TUGboat

Volume 33, Number 1 / 2012

TEX Users Group

TUGboat (ISSN 0896-3207) is published by the TEX Users Group.

Memberships and Subscriptions

2012 dues for individual members are as follows:

- Ordinary members: $95.
- Students/Seniors: $65.

The discounted rate of $65 is also available to citizens of countries with modest economies, as detailed on our web site.

Membership in the TEX Users Group is for the calendar year, and includes all issues of *TUGboat* for the year in which membership begins or is renewed, as well as software distributions and other benefits. Individual membership is open only to named individuals, and carries with it such rights and responsibilities as voting in TUG elections. For membership information, visit the TUG web site.

Also, (non-voting) *TUGboat* subscriptions are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. The subscription rate is $100 per year, including air mail delivery.

Institutional Membership

Institutional membership is a means of showing continuing interest in and support for both TEX and the TEX Users Group, as well as providing a discounted group rate and other benefits. For further information, see `http://tug.org/instmem.html` or contact the TUG office.

Trademarks

Many trademarked names appear in the pages of *TUGboat*. If there is any question about whether a name is or is not a trademark, prudence dictates that it should be treated as if it is. The following trademarks which commonly appear in *TUGboat* should not be considered complete.

TEX is a trademark of American Mathematical Society.
METAFONT is a trademark of Addison-Wesley Inc.
PostScript is a trademark of Adobe Systems, Inc.

[printing date: April 2012]

Printed in U.S.A.

Addresses

TEX Users Group
P. O. Box 2311
Portland, OR 97208-2311
U.S.A.

Telephone

+1 503 223-9994

Fax

+1 206 203-3960

Web

`http://tug.org/`
`http://tug.org/TUGboat/`

Electronic Mail
(Internet)

General correspondence, membership, subscriptions:
`office@tug.org`

Submissions to *TUGboat*, letters to the Editor:
`TUGboat@tug.org`

Technical support for TEX users:
`support@tug.org`

Contact the Board of Directors:
`board@tug.org`

[. . .] clay possesses one considerable advantage: it is resistant to fire, water, and magnetic disturbances. In [. . .] a few thousand years, our photographs, books, and hard disks will no doubt have disappeared, but our collections of cuneiform tablets will still be there.

Dominique Charpin, *Reading and Writing in Babylon* (2010)

translated by Jane Marie Todd

TUGBOAT

COMMUNICATIONS OF THE TEX USERS GROUP
EDITOR BARBARA BEETON

VOLUME 33, NUMBER 1 • 2012
PORTLAND • OREGON • U.S.A.

TUG 2012

The 33rd Annual Meeting of the TEX Users Group
Presentations covering the TEX world

July 16–18, 2012 ▪ Boston, Massachusetts, USA

http://tug.org/tug2012 ▪ tug2012@tug.org

April 30, 2012 — bursary application deadline
May 1, 2012 — presentation proposal deadline
May 15, 2012 — early bird registration deadline
June 1, 2012 — preprint submission deadline
July 16, 2012 — LaTeX workshop (concurrent)
July 16–18, 2012 — conference
July 30, 2012 — deadline for final papers

Sponsored by the TEX Users Group and DANTE e.V.

Ab Epistulis

Steve Peter

Hello from the TEX world headquarters! I'm pleased to be able to share with you the following information from the TEX Users Group, my first as president. (And I'm pining for the days when Karl Berry wrote these messages.)

Group membership category

This year, TUG is offering a new membership category, called group membership. It includes up to four individual memberships for $200 (thus providing a discount), with electronic access for all four members and one physical copy of *TUGboat* and software, as well as an acknowledgment online. This is perfect for small departments or research groups. http://tug.org/join.html has all the info.

If you haven't joined TUG yet for this year, it's not too late. Individual memberships are just $95 and joining TUG helps to support and promote the use of TEX, LATEX, ConTEXt, METAFONT, METAPOST, and related systems worldwide.

Software

No major changes in the software delivery are expected this year. Work toward the TEX Live 2012 release is well underway. We will begin trial builds soon and are aiming to freeze updates in May (a timeline is on http://tug.org/texlive/). The editors of the MacTEX, proTEXt, and CTAN components of the overall TEX Collection software are also preparing their respective releases.

Conferences

TUG 2012 will be held in Boston, Massachusetts, USA, from July 16 through July 18 at the Omni Parker House. The deadline for presentation abstracts is May 1, 2012, and early bird registration is available through May 15. Registration includes breakfast, lunch, and coffee breaks as well as all the cutting edge TEX information that will fit in your brain. The discount code for our group's hotel reservations is available on the conference web site at http://tug.org/tug2012/.

Outside of North America, GUST is celebrating its 20th anniversary with a special BachoTEX from April 29 through May 3. For all the information, see the website at http://www.gust.org.pl/bachotex/2012/.

TEXperience 2012 (CSTUG) will be held August 23–26 in Morávka, Czech Republic. See the website at http://katedry.osu.cz/kma/TeXperience2012 for all the details.

EuroTEX 2012 and the Sixth ConTEXt User Meeting will be October 8–12 in Breskens, The Netherlands. See http://meeting.contextgarden.net/2012/.

Book reviews

The TUG website has a section for book reviews, and you should definitely visit if you haven't yet had the chance. Recently, we've added reviews of David Kadavy's *Design for hackers: Reverse-engineering beauty* (reviewed by Boris Veytsman) and Jean-luc Doumont's *Trees, maps, and theorems* (by Pavneet Arora). These and other reviews are at http://tug.org/books/#reviews. Many thanks to Boris Veytsman for organizing the reviews.

And speaking of books, RIT Press/RIT Cary Graphic Arts Press is offering 10% off to TUG members for a limited time. The code may be used on one order and is valid through May 31, 2012. The exclusive members-only discount code is available in the members area at https://www.tug.org/members.

◇ Steve Peter
president (at) tug dot org
http://tug.org/TUGboat/Pres/

Editorial comments

Barbara Beeton

Don Knuth, reprise

To cap off his publications so far, and leave his time free to work on *TAOCP*, Don has created one final volume, a *Companion* to his collected works, which is reviewed elsewhere in this issue.

An addition to Don's online biographical entries now appears on the new Turing Award web site: http://amturing.acm.org/award_winners/knuth_1013846.cfm. The information was compiled and the entry written by Dave Walden, ringleader of the TUG Interview Corner.

An alternate view of CTAN

The "Automated Mercurial Repositories of CTAN" web site http://ctanhg.scharrer-online.de provides an archive of old versions of CTAN packages. As well as allowing (LA)TEX historians to follow the development of packages, it provides an emergency backup in case a package change prevents an old document from being processed properly.

Linotype: The Film

The Linotype, invented by Ottmar Mergenthaler in the late 1800s, was the machine that, for all practical

purposes, put hand compositors out of business. On it were composed uncounted books, magazines, job work, and newspapers through the mid-20th century. Then "cold type" and photocomposition came into existence, followed by digital print and the desktop revolution, and the world's Linotypes fell silent.

This fascinating machine has now been memorialized in a film. Released early this year, the film premiered in New York, and a screening followed soon after at the Rhode Island School of Design, which I was privileged to attend. It's an engaging story, told well with great empathy and appreciation.

More screenings will continue, and a DVD is expected in early summer. Visit the web site (`http://www.linotypefilm.com`), view the trailer, check the schedule, and see it if you can.

Barriers to effective communication: Jean-luc Doumont

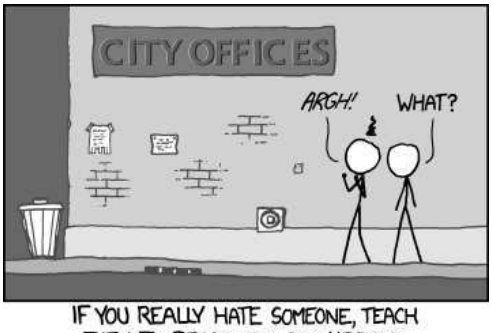
As an extension to his presentation at TUG 2011, Jean-luc Doumont has prepared a booklet setting forth the many reasons that authors hesitate, even actively resist, enhancing the presentation of their work — "we can't do it this way," .... This tale of woe is followed by Jean-luc's reasons why it *can* be done, and how to persuade the powers that be to allow it to happen.

The booklet can be downloaded from `http://www.treesmapsandtheorems.com/barriers`.

Kern it!

An interactive game, "KERNTYPE" (`http://type.method.ac`), instructs a visitor to "kern me", and invites one to drag letters around horizontally to see how well the results compare with a typographer's solution. It provides both amusement and good feedback in tuning one's sensitivity to this nicety.

On the other hand, overly sensitive attention to this detail can provide both puzzlement and frustration, as illustrated by this cartoon.[1]



IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

[1] Source: `http://xkcd.com/1015/`; posted there under Creative Commons Attribution-NonCommercial 2.5 License, and used with thanks and hilarious appreciation.

An advertising circular caught my eye. In the headline was the first instance I can remember where the kerning of "AY" looked too tight. All caps, the phrase "DOLLAR DAYS" presents significant kerning problems.

Granted, it's not the most elegant font, but it's an ad, after all. At least it's possible to tell that it's two words...

One last word on the subject: `http://school.failblog.org/2012/03/27/` (but not the first word on the page...).

A wonderful use of old books

As much as I hate to see a book destroyed, evidence of inspired artistic license merits forgiveness. Throughout most of 2011, the "library phantom" made the rounds of Edinburgh libraries and similar venues, leaving behind the most enchanting paper sculptures — all anonymous — "... in support of libraries, books, words, ideas ...". Here is a report by Robert Krulwich on the National Public Radio web site: `http://www.npr.org/blogs/krulwich/2011/10/28/141795907`. Follow the links — *all* of them!

The Plantin-Moretus Museum in Antwerp

The Plantin-Moretus Museum (a wonderful place! Both the city and the museum) now has an English language web site: `http://museum.antwerpen.be/plantin_moretus/index_eng.html`.

The oldest part of the buildings that now house the museum was established by Christoffel Plantin as a printing house in 1555. It remained in continuous operation, managed by the Moretus family (the first of whom was Plantin's son-in-law), until the 1820s. In 1876, after several unsuccessful attempts to revive the business, Edward Moretus sold the buildings and their entire contents to the city of Antwerp; in 1877 both the living quarters and the printing office were opened to the public as a museum. The collection includes publications from the earliest days of the firm, as well as two of the oldest printing presses in the world, and much of the original type, still used to develop authoritative digitized versions. Definitely worth a visit!

◇ Barbara Beeton
`http://tug.org/TUGboat`
`tugboat (at) tug dot org`

Hyphenation exception log

Barbara Beeton

This is the periodic update of the list of words that TEX fails to hyphenate properly. The full list last appeared in *TUGboat* 16:1, starting on page 12, with updates in *TUGboat* 22:1/2, pp. 31–32; 23:3/4, pp. 247–248; 26:1, pp. 5–6; 29:2, p. 239, and 31:3, p. 160.

In the list below, the first column gives results from plain TEX's `\showhyphens{...}`. The entries in the second column are suitable for inclusion in a `\hyphenation{...}` list.

In most instances, inflected forms are not shown for nouns and verbs; note that all forms must be specified in a `\hyphenation{...}` list if they occur in your document. The full list of exceptions, as a TEX-readable file, appears at `http://mirror.ctan.org/info/digests/tugboat/ushyphex.tex`. (It's created by Werner Lemberg's scripts, available in the subdirectory `hyphenex`.)

Like the full list, this update is in two parts: English words, and names and non-English words (including transliterations from Cyrillic and other non-Latin scripts) that occur in English texts.

Thanks to all who have submitted entries to the list. Here is a short reminder of the relevant idiosyncrasies of TEX's hyphenation. Hyphens will not be inserted before the number of letters specified by `\lefthyphenmin`, nor after the number of letters specified by `\righthyphenmin`. For U.S. English, `\lefthyphenmin=2` and `\righthyphenmin=3`; thus no word shorter than five letters will be hyphenated. (For the details, see *The TEXbook*, page 454.) This particular rule is violated in some of the words listed; however, if a word is hyphenated correctly by TEX except for "missing" hyphens at the beginning or end, it has not been included here.

Some other permissible hyphens have been omitted for reasons of style or clarity. While this is at least partly a matter of personal taste, an author should think of the reader when deciding whether or not to permit just one more break-point in some obscure or confusing word. There really are times when a bit of rewriting is preferable.

One other warning: Some words can be more than one part of speech, depending on context, and have different hyphenations; for example, 'analyses' can be either a verb or a plural noun. If such a word appears in this list, hyphens are shown only for the portions of the word that would be hyphenated in the same way regardless of usage.

The reference used to check these hyphenations is *Webster's Third New International Dictionary*, unabridged.

Hyphenation for languages other than U.S. English

Patterns now exist for many languages other than U.S. English, including languages using accented and non-Latin alphabets. CTAN holds an extensive collection of patterns: see `language/hyphenation` and its subdirectories.

A group of volunteers led by Mojca Miklavec and Manuel Pégourié-Gonnard have created a comprehensive package of hyphenation patterns, called `hyph-utf8`; see `http://tug.org/tex-hyphen`.

The list — English words

`acronym`	acro-nym
`anachro-nism(tic)`	anach-ro-nism(-tic)
`anal-y-sis`	analy-sis
`ap-pen-dices`	ap-pen-di-ces
`ap-pendix`	ap-pen-dix
`au-tore-gres-sion`	auto-re-gres-sion
`au-tore-gres-sive`	auto-re-gres-sive
`bedrag-gle`	be-drag-gle
`bedrock`	bed-rock
`bed-warf`	be-dwarf
`bib-li-o-graph-i-cal`	bib-lio-graph-i-cal
`bi-bunits`	bib-units
`bioweapon(s,ry)`	bio-weap-ons(-ry)
`bungee`	bun-gee
`cochlea(s,r)`	coch-leas(r)
`code-signer`	co-designer
`con-geries`	con-ge-ries
`cosemisim-ple`	co-semi-sim-ple
`cy-bervirus(es)`	cy-ber-virus(es)
`cy-ber-weapon`	cy-ber-wea-pon
`dis-tributable`	dis-trib-ut-able
`dis-tribu-tive`	dis-trib-u-tive
`ecosys-tem`	eco-sys-tem
`economies`	econ-o-mies
`en-do-scopies`	en-dos-copies
`en-doscopy`	en-dos-copy
`eu-stachian`	eu-sta-chian
`flu-o-ro-scopies`	fluor-os-copies
`flu-o-roscopy`	fluor-os-copy
`geode-tic`	ge-o-det-ic
`grou-p-like`	group-like
`halflife(ves)`	half-life(ves)
`he-liopause`	he-lio-pause
`he-liotrope`	he-lio-trope
`holodeck`	holo-deck
`hound-steeth`	hounds-teeth
`hound-stooth`	hounds-tooth
`hy-per-e-las-tic-ity`	hy-per-elas-tic-ity
`hy-poe-las-tic-ity`	hy-po-elas-tic-ity
`illiq-uid(ity)`	il-li-quid(-ity)
`let-terspace(s,d)`	let-ter-spaces(d)

liq-uid-ity	li-quid-ity
looka-head	look-ahead
macroe-con-omy	macro-econ-omy
megafauna(l)	mega-fau-na(l)
metasta-bil-ity	meta-sta-bil-ity
metastable	meta-stable
meta-table	meta-table
meta-tables	meta-tables
method	meth-od
mi-croe-con-omy	micro-econ-omy
mi-croen-ter-prise	micro-en-ter-prise
mi-crostruc-ture	mi-cro-struc-ture
monospac-ing	mono-spacing
pager-ank	page-rank
plateau	pla-teau
purges	pur-ges
reed-u-cate	re-edu-cate
refugee	ref-u-gee
satel-lite	sat-el-lite
sha-pable	shap-able
sin-glespace(d)	single-space(d)
sin-glespac-ing	single-spacing
sl-nuni-code	sln-uni-code
spokesman	spokes-man
spokesper-son	spokes-per-son
sub-tables	sub-tables
su-perderiva-tion	super-deri-va-tion
surgery	sur-gery
surg-eries	sur-ge-ries
surges	sur-ges
takeover	take-over
topoi-so-merase	topo-iso-mer-ase
weapon(s,ry)	weap-ons(-ry)

Names and non-English words

Apol-lodorus	Apol-lo-dorus
Be-bchuk	Beb-chuk
Bur-ck-hardt	Burck-hardt
Chester	Ches-ter
Chi-ang	Chiang
Chich-ester	Chich-es-ter
Co-hen	Cohen
Dor-fleit-ner	Dorf-leit-ner
Drech-sler	Drechs-ler
Ei-jkhout	Eijk-hout
En-gle	Engle
En-gel	Engel
Gesellschaft	Ge-sell-schaft
Got-tlieb	Gott-lieb
Hu-ber	Huber
Jun-gian	Jung-ian
Key-ne-sian	Keynes-ian
Kro-necker	Kron-ecker
Lu-cas	Lucas
Mac-Beth	MacBeth
Mag-el-lan	Ma-gel-lan
Methodist	Meth-od-ist
Method-ism	Meth-od-ism
No-towidigdo	Noto-wi-digdo
Ob-st-feld	Obst-feld
Ore-opou-los	Oreo-pou-los
Raviku-mar	Ravi-kumar
Re-ich-lin	Reich-lin
Schim-melpfen-nig	Schim-mel-pfen-nig
Schw-ert	Schwert
Thiru-vanan-da-pu-ram	Thiruv-ananda-puram
Toy-ota	Toyo-ta
We-in-stein	Wein-stein
William(s)	Will-iam(s)
Wolf-fian	Wolff-ian

TUGboat editorial information

This regular issue (Vol. 33, No. 1) is the first issue of the 2012 volume year. No. 2 will contain papers from the TUG 2012 conference in Boston, Massachusetts, USA, and no. 3 will be a regular issue.

TUGboat is distributed as a benefit of membership to all current TUG members. It is also available to non-members in printed form through the TUG store (`http://tug.org/store`), and online at the *TUGboat* web site, `http://tug.org/TUGboat`. Online publication to non-members is delayed up to one year after print publication, to give members the benefit of early access.

Submissions to *TUGboat* are reviewed by volunteers and checked by the Editor before publication. However, the authors are still assumed to be the experts. Questions regarding content or accuracy should therefore be directed to the authors, with an information copy to the Editor.

Submitting items for publication

The deadline for receipt of final papers for the next issue is July 30, and for the one after, October 1.

Suggestions and proposals for *TUGboat* articles are gratefully accepted. Please submit contributions by electronic mail to `TUGboat@tug.org`.

The *TUGboat* style files, for use with `plain` TEX and LATEX, are available from CTAN and the *TUGboat* web site. We also accept submissions using ConTEXt. More information and tips for authors are at:
`http://tug.org/TUGboat/location.html`

Effective with the 2005 volume year, submission of a new manuscript implies permission to publish the article, if accepted, on the *TUGboat* web site, as well as in print. Thus, the physical address you provide in the manuscript will also be available online. If you have any reservations about posting online, please notify the editors at the time of submission and we will be happy to make special arrangements.

TUGboat editorial board

Barbara Beeton, *Editor-in-Chief*
Karl Berry, *Production Manager*
Boris Veytsman, *Associate Editor, Book Reviews*

Production team

William Adams, Barbara Beeton, Karl Berry, Kaja Christiansen, Robin Fairbairns, Robin Laakso, Steve Peter, Michael Sofka, Christina Thiele

TUGboat advertising

For advertising rates and information, including consultant listings, contact the TUG office, or see:
`http://tug.org/TUGboat/advertising.html`

In memoriam: Tony Siegman, 1931–2011

Bruce Armbruster and
Jeannie Howard Siegman

Anthony E. Siegman, laser pioneer and professor emeritus of electrical engineering and applied physics at Stanford, died at home on October 7, 2011. His 1,283 page text, *Lasers*, published in 1986, was one of the first major science textbooks published from start to finish using TEX and instantly became a classic. Its acceptance and success established that TEX was capable of producing books that were just as elegant and attractive as traditionally typeset books, at a fraction of the cost, and free of errors introduced by conventional production processes. The publication of *Lasers* helped change forever the way that scientific and technical books are published and Tony Siegman played a critical role in that revolution.

A Michigan native, Tony completed his AB degree Summa Cum Laude in three years as a National Merit Scholar at Harvard, where he played the clarinet in the Harvard Marching Band. After two years on a cooperative plan with UCLA and the Hughes Research Labs in Culver City, California, he moved north to Stanford. There he was appointed to the Stanford faculty on an acting basis in 1956, and received his PhD degree in Electrical Engineering in 1957 with a dissertation on microwave noise in electron beams and traveling-wave tubes.

Tony was part of the program committee and an active participant in the historic first Quantum Electronics symposium at Shawanga Lodge, New York, in 1959. That marked the start of serious research into lasers. Thereafter, he rapidly began to move his research from microwaves and masers to optics and lasers. After 1960, his work evolved into a long research and teaching career in lasers and optics, during which he supervised some 40 PhD dissertations and published numerous scientific articles and three textbooks: *Microwave Solid-State Masers* (McGraw-Hill, 1964), *An Introduction to Lasers and Masers* (McGraw-Hill, 1972), and *Lasers* (University Science Books, 1986).

Tony's foremost technical contribution is probably his invention of the unstable resonator — a conceptual advance that made possible high-power lasers with high beam quality. He directed the Ginzton Laboratory at Stanford from 1978 to 1983 and again in 1998–99, and served on numerous academic committees and as a member of the Stanford Faculty Senate and its Steering Committee. He spent sabbaticals as Visiting Professor of Applied Physics at Harvard in 1965, Guggenheim Fellow at the IBM Research Labs in Zurich in 1969–70, and Humboldt Senior Scientist at the Max Planck Institute for Quantum Optics in Garching, Germany, in 1984–85.

He was regarded by many as a true patriarch in his field, and remembered with warmth and admiration by his students and colleagues alike. His professional colleagues have initiated an endowment fund to carry on, and now named in his honor, the Siegman International Summer Session on Lasers and Their Applications. The prototype in 2011 was the last of his life-long professional volunteer activities. For more about the project, or to make a contribution in his memory, please see `http://www.osa-foundation.org/Siegman`.

◇ Bruce Armbruster
University Science Books

◇ Jeannie Howard Siegman
Stanford University Staff Emerita

Typographers' Inn

Peter Flynn

1 Titling and centering

I forget who first pointed it out, but one of the earliest pieces of advice I remember was to break title lines according to sense when they are set centered. Anyone who wants to glance at their organization's noticeboards can see the effect of not knowing this: how often do you see isolated words on lines by themselves, as if they were an afterthought.

Breaking according to sense means reading the title and seeing where the natural pauses in rhythm and meaning occur, and making breaks there. Failing to do this not only means the title is harder to read, because the brain has to struggle to join back together phrases or words which should not have been split up, but also, depending on the way it has been broken, you can sometimes even end up with a humorous misinterpretation.

Many people will be familiar with this phrase

PARIS
IN THE
THE SPRING

in which 'the' is duplicated. That is deliberate, as was the poster a colleague at the London College of Printing designed for an exhibition called 'Things aren't what they seem to be'. The opportunity was too good to pass up (Figure 1).

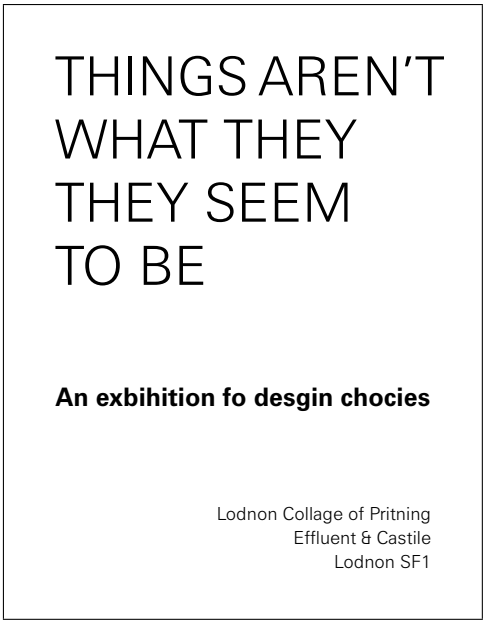


Figure 1: Student poster, c.1975, author's reconstruction from memory

Alas, I can only reconstruct a rudimentary copy here. For those unfamiliar with London, the Elephant and Castle was then a rather run-down district on the south bank, still showing the scars of WWII bombing and the ghastly New Barbarism of the 50s and 60s reconstruction.

More recently my local convenience store was advertising

HALF-PRICE
DESSERTS FROM
OUR IN-STORE
BAKERY

It's only ephemeral, but it would have been more readable, especially to passers-by, if it had read

HALF-PRICE DESSERTS
FROM OUR
IN-STORE BAKERY

Closely related to this is a combination of unfortunate circumstances which I first saw on the spine of a book. The front cover used the same face for author and title, but in different colors: a mauve for the author and a washed-denim for the title (Figure 2).

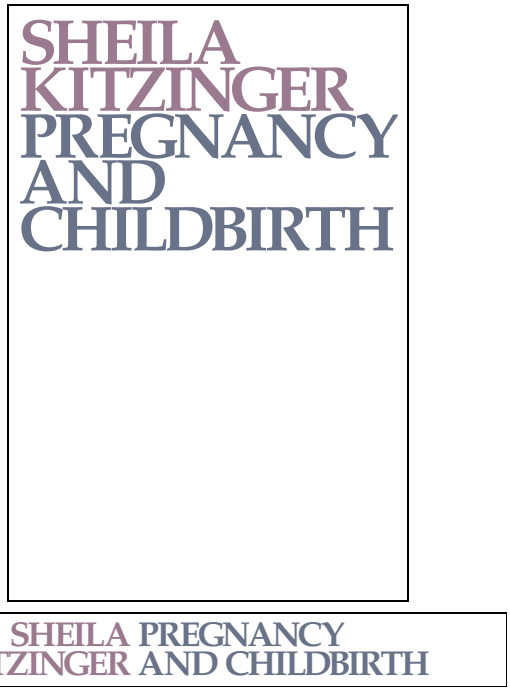


Figure 2: Front cover and spine (reconstruction, omitting detail)

The spine used the same colors and typeface, but arranged with the author name to the left, two lines flushright, hard up against the title to the right, two lines flushleft. The unfortunate combination of the close spacing and the fading of the colors over

time leaves one with the impression that this was Sheila Pregnancy's *Kitzinger and Childbirth*.

A close contender is the copy of Stieg Larsson's *The Girl with the Dragon Tattoo* which I was given for Yuletide. The spine almost leaves you with the impression that is it *THE GIRL WITH THE STIEG* by DRAGON TATTOO LARSSON.

2 Beaten into submission

Some years ago I signed up to do a late-life part-time PhD. It's been a long time a-growing, but it's getting somewhere near fruition. Which is all well and good, and nothing to do with this column, except that as a by-product, I was obviously going to produce the thing using LaTeX. My university didn't even have a Word template for theses, let alone a thesis document class for LaTeX, so it equally obviously behove me to write one.

I made a start, initially for my own benefit, but as this was a part-time task slaved off another part-time task slaved off a full-time task, it tended to get quietly ignored while I fiddled with more burning issues like research and a day job. My patient and long-suffering co-students were repeatedly told 'it's on its way', but it took until last summer for something usable to appear. I started with the crufty and encrusted code that I had been working with, chipped off the barnacles, oiled the insides, fixed the broken bits, added the requisite bells and whistles, polished the outsides — and then had to start thinking about appearances.

The university's regulations are simple to the point of being simplistic: one-sided; a 4cm left-hand (binding) margin, 3cm everywhere else; a 'plain' typeface (that is, not ornamental or decorated); the text double-spaced or $1\frac{1}{2}$-line-spaced; some specific items on the (centred) title page; a table of contents, Declaration, and 200–word Abstract. That's about it, so the detail was basically up for grabs by each student. The Graduate Studies Office, which deals with thesis submission and reception, do in fact impose other rules, but these are so obvious and so rarely broken that they are not usually mentioned (like it has to be printed in black on white paper).

I'm not a graphic designer nor a professional typographer nor even a typographic designer, but I am aware that working to a very loose brief is in some ways harder than working to one too tight. It's also axiomatic that simplicity is best when dealing with a functional document whose primary purposes are to act as a record of your findings and to convince your Extern that you are safe to let loose on the world of research. However, the more I looked at the largely pragmatic and temporary choices I had made for my drafts, the more I realised the dangers of allowing the technology to dominate the design: just because LaTeX (or whatever system) *can* do something, doesn't mean that you *have* to do it. One of the pitfalls of having spent a lifetime working with computer documentation is that there is a tendency for all document content to end up looking like *The TeXbook*, regardless of the page layout and the typeface.

As it turned out, I made a lot of cuts and very few additions. I had in fact just been experimenting with different layouts for floats before deciding that the marginal improvement in balance between float and the surrounding text wasn't worth the additional weight of code, so I carried this principle over to the rest of the class.

The document class loads the default report with the options `oneside,11pt,a4paper`, thus the only variations are those required by the regulations, and a few which I have seen fit to impose on myself. There are discussions about printing two-sided to save trees, but with electronic submission imminent, the argument appears to be moot.

Layout: a separate title page and one-sided imposition are mandated; A4 is the only paper size, although it will allow Letter, in case students with US funding agencies or sponsors need a copy to fit their filing system. It was not hard to fix the text area so that it works acceptably if not optimally on both sizes of paper.

Type size: currently fixed at 11pt — I would have preferred 12pt, because the important people reading it (supervisor, Prof, Extern) tend to be much older than the student, and therefore have suboptimal eyesight, but 12pt does increase the page count significantly. I need to experiment with the balance between line-spacing and type size for this exact line-length.

Line-spacing: double-spacing makes the thesis far too many pages, although it has a legitimate purpose for drafts, where readers appreciate having the space to scribble comments. Line-and-a-half looks about right, but there's nothing to stop the student overriding this. A 10pt setting is clearly too small for the length of lines on an office-size sheet of paper with the given margins without resorting to double line-spacing or more.

Typeface: for those including a significant amount of mathematics, the scope is very limited, and I suspect most will stick to CM or Times. Otherwise it's up to the author, within the confines of the regulations. I used Charter, because I find it reads easily in very long documents.

Justification: I made the whole document default to `\raggedright` for several reasons: a PhD thesis is not a book for printing to publishers' standards; it's not good for an author to spend a lot of time fiddling with manual reformatting to 'make things fit'; and the line-length is already too wide for comfortable reading. It's not an ideal choice, so I am providing an option to switch back to justified setting for those who feel they can deal with it. I posted about this on `comp.text.tex` while writing this column, so I'm still open to suggestions.

Paragraph spacing: The default uses the parskip package, so there is no indentation, and about one baseline's space between paragraphs. I did provide an option to switch back to indented, unspaced paragraphs at the insistence of some users who wanted a more book-like layout.

The bulk of the class file is taken up with options for the preselection of the School or Department and the class of Degree using abbreviations as options, so that the title page can be composed correctly. The code list is refreshed annually from the Registrar's database of the graduate courses on offer, so authors still finishing degrees in disciplines no longer on offer to new students, or in schools or departments no longer operating solo, have to be catered for by preserving the validity of old options.

The design of the title page is straightforward, and largely mandated by the prescribed contents: title, author, department, degree, supervisor, date, and so on, in varying sizes, and centred.

Some things in the text body, however, I did change, simply because the default LaTeX layout naturally reflects the typographic habits of US book design of the 1970s and 80s, and I feel it is time to move on.

Front matter: Apart from providing environments for the Dedication and Acknowledgments, there is a generic `\prelim` command for optional preliminary sections like admonishments, standards certification, and terminologies, not part of the thesis proper. It is basically a `\section*` command, except that it also adds itself to the ToC.

For the ToC, LoF, LoT, Bibliography (and Index, if anyone wants one), the heading is at section level, starting a new page, not (by default) at chapter level. These are not divisions of chapter status, perhaps excepting the Bibliography, and it is usually wrong to accord them that format.

The presence of a LoF and LoT is automated, using a Boolean switch added to the `.aux` file at the end if either the `figure` or `table` counter is non-zero. This can be overridden if, for example, an author has only a very few figures or tables.

Description lists: LaTeX's default, as with this list, is to format the topic heading run-in to the text. This is idiosyncratic, and nowadays unusual outside the dictionary format. The separate-line format, as exemplified in HTML's `<dt>` element type seems to be expected by readers and writers alike, so I changed the `\descriptionlabel` to use a `\parbox` to give the topic heading a line or lines to itself. This also overcomes the standing bug in LaTeX whereby very long labels fail to break correctly at the end of a line.

Quotations: The quotation environment now has an optional argument for the key value for the citation of the quote, which is printed flush right after the text. A similarly-implemented epigraph environment does the same for authors wanting introductory quotes for their chapters.

So all in all, I hope it stays a commonplace and unobtrusive layout, which is how a thesis design should be: it's the student's arguments that should stand out, not the typography. It's available within the university on pilot, and will be released into the wild once I'm satisfied that it's as student-proof as needed.

Currently I am still recommending BibTeX to users: we have a lot of them still depending on old `.bst` files, and until I can get to deal with these, I'm holding back on recommending *biblatex*.

Afterthought

I need to update my online book, *Formatting Information* to reflect TeX Live 2011 and a lot of package updates. There are also some substantial parts that have been superseded by technology — it doesn't need a whole chapter on how to print from a DVI or PS file any more.

If you have read or used it in teaching, I'd be grateful for suggestions for improvement (`http://latex.silmaril.ie/formattinginformation`).

And just as we go to press, I have started to see badly-broken display lines in flush-left text, not just in centred text. Grrr.

◇ Peter Flynn
Textual Therapy Division, Silmaril Consultants
Cork, Ireland
Phone: +353 86 824 5333
`peter (at) silmaril dot ie`
`http://blogs.silmaril.ie/peter`

Lucida OpenType fonts available from TUG

Karl Berry

The TEX Users Group is happy to announce the availability of a new incarnation of the Lucida typeface family: Lucida OpenType. The web pages with ordering information, samples, documentation, and more are at `http://tug.org/lucida`. The previous Lucida Type 1 distribution is still available as well.

Usage

The Lucida OpenType fonts can be used with the TEX engines that support OpenType, namely XETEX and LuaTEX, and with any other OpenType-aware application or system, such as LibreOffice/OpenOffice and Microsoft Word.

The Lucida OpenType distribution contains the following text fonts:

- `LucidaBrightOT` in the usual four variants: regular, italic, bold, bold italic;
- `LucidaSansTypewriterOT` (same four variants);
- `LucidaSansOT` (same four variants);
- and three special fonts: `LucidaBlackletterOT`, `LucidaCalligraphyOT`, `LucidaHandwritingOT`.

The first three of these font sets have been considerably extended and revised compared to the Type 1 distribution; for example, eastern European languages are generally supported now, and the math font includes a newly-designed script alphabet. The special fonts in the last item have simply been carried over from the Type 1 distribution.

In addition, there are two math fonts:

- `LucidaBrightMathOT`, a full OpenType math font, also extended and revised compared to the Type 1 Lucida math support. OpenType features are available to switch between some alternative glyph designs.
- `LucidaBrightMathOT-Demi`, a variant with bold letters in the normal positions, to be used for typesetting math within bold text, such as section headings. This is distinct from the math characters which are specified as bold in Unicode for specific semantics; those are bold in both math fonts.

A previous article [1] describes the context, history, and some technical aspects of the project.

Availability

The online order form for both Lucida OpenType and Lucida Type 1 is at `http://tug.org/lucida`. The pricing is the same for both distributions. Both individual and site licenses are available. There is a substantial discount for members of TUG (and other TEX user groups).

The Lucida distributions are released over the Internet only; there are no physical CDs. Lucida OpenType is available in either of two zip files: one has subdirectories arranged according to the TEX Directory Structure (`http://tug.org/tds`), the other is a simple flat archive for those who have no need of TEX directories. Both contain exactly the same files.

The Type 1 fonts are frozen and will not be developed further, while the OpenType fonts continue to be actively maintained and developed.

Credits

TUG gratefully acknowledges Charles Bigelow and Kris Holmes for their enthusiasm in enhancing their magnum opus, and Khaled Hosny for doing the technical work. Mojca Miklavec and Hans Hagen originally promulgated the project. Thanks to them and to additional volunteers for testing, suggestions, and advice: Taco Hoekwater, Bogusław Jackowski, Will Robertson, Michael Sharpe, and Ulrik Vieth. And many thanks to our stalwart beta testers and reviewers: Barbara Beeton and Stephen Moye at the AMS, Axel Retif, and Steve Peter.

Samples

ABC xyz JKL 0248 LucidaBrightOT
ABC xyz 0248 *LucidaBrightOT-Italic*
ABC xyz JKL 0248 **LucidaBrightOT-Demi**
ABC xyz 0248 ***LucidaBrightOT-DemiItalic***

ABC xyz LucidaSansTypewriterOT
ABC xyz *LucidaSansTypewriterOT-Oblique*
ABC xyz **LucidaSansTypewriterOT-Bold**
ABC xyz ***LucidaSansTypewriterOT-BoldOblique***

ABC xyz LucidaSansOT
ABC xyz *LucidaSansOT-Italic*
ABC xyz **LucidaSansOT-Demi**
ABC xyz ***LucidaSansOT-DemiItalic***

𝔄𝔅ℭ xyz 𝔏𝔲𝔠𝔦𝔡𝔞𝔅𝔩𝔞𝔠𝔨𝔩𝔢𝔱𝔱𝔢𝔯𝔒𝔗
𝒜ℬ𝒞 xyz *LucidaCalligraphyOT*
ABC xyz *LucidaHandwritingOT*

$e^{i\pi} + 1 = 0$ LucidaBrightMathOT
$\boldsymbol{e^{i\pi} + 1 = 0}$ LucidaBrightMathOT-Demi

References

[1] Ulrik Vieth and Mojca Miklavec, Another incarnation of Lucida: Towards Lucida OpenType. *TUGboat* 32:2, pp. 169–176, 2011. `tug.org/TUGboat/32-2/tb101vieth.pdf`.

◇ Karl Berry
`http://tug.org/lucida`

The Amiri typeface

Khaled Hosny

In 1905, the famous Bulaq printing press in Cairo (also known as al-Amiriya, the royal press) issued a new Arabic typeface as part of reviving the then moribund printing press. This typeface later came to be one of the most widely used and highly regarded Arabic typefaces, even in the digital era.

Arabic has strong calligraphic traditions with many styles, and Naskh ("to copy") is the most commonly used style in typesetting. One of the most novel features of the Bulaq typeface is maintaining the æsthetics of Naskh calligraphy while meeting the requirements (and limitations) of typesetting, a balance that is not easily achieved.

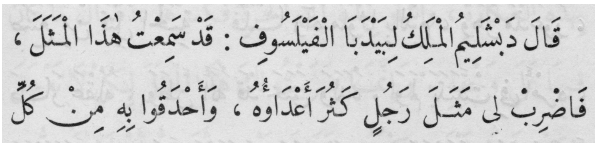
Amiri is a revival of that typeface, and though it is not the first one, I believe it is the most elaborate and most complete, as all other revivals omit many of the letter forms in the metal type either for simplicity or limitations of early digital systems. On the other hand, features that are merely a result of the limitations of metal typesetting are not reproduced in Amiri, when appropriate.

Work on the Amiri typeface started slowly in late 2008, with the first alpha release in November 2010 and the first beta in December 2011. Though formally still in beta stage, it is now considered to be mature enough for general use. It will not be marked stable until there are no metric-incompatible changes planned.

The Amiri family includes regular, bold, slanted and bold slanted fonts. Though slanted type is not a particularly Arabic concept, it is provided because of widespread use in contemporary typesetting, especially on the web, and right-leaning fake-slanted Arabic is very unnatural. The bold font is not as polished as the regular one, and still needs more work.

Amiri fully covers the "Arabic" and "Arabic Supplement" blocks in version 6.0 of Unicode, and thus it supports any language written in Arabic that is supported by Unicode. This includes, for example, Arabic, Fula, Hausa, Jawi, Kashmiri, Kurdish, Ottoman Turkish, Pashto, Persian, Punjabi, Sindhi, Swahili, Urdu, Uyghur and Wolof. Work on new Arabic additions to version 6.1 of the standard is under way. "Arabic Presentation Forms–A" and "Arabic Presentation Forms–B" blocks are also covered for the sake of completeness, though they are composed mostly of compatibility characters.

Great care has been taken to make sure Amiri can be used to typeset the Qur'an (the book of Islam)


قَالَ دَبْشَلِيمُ الْمَلِكُ لِبَيْدَبَا الْفَيْلَسُوفِ : قَدْ سَمِعْتُ هٰذَا الْمَثَلَ ،
فَاضْرِبْ لِي مَثَلَ رَجُلٍ كَثُرَ أَعْدَاؤُهُ ، وَأَحْدَقُوا بِهِ مِنْ كُلِّ

Sample from Kalilah wa Dimnah, Bulaq, 1938.

أبجد هوز حطي كلمن سعفص قرشت ثخذ ضظغ ٠١٢٣٤٥٦٧٨٩٪
أبجد هوز حطي كلمن سعفص قرشت ثخذ ضظغ ٠١٢٣٤٥٦٧٨٩٪
أبجد هوز حطي كلمن سعفص قرشت ثخذ ضظغ ٠١٢٣٤٥٦٧٨٩٪
أبجد هوز حطي كلمن سعفص قرشت ثخذ ضظغ ٠١٢٣٤٥٦٧٨٩٪

The four styles in the Amiri family.

صِفْ خَلْقَ خَوْدٍ كَمِثْلِ ٱلشَّمْسِ إِذْ بَزَغَتْ
يَحْظَى ٱلضَّجِيعُ بِهَا نَجْلَاءَ مِعْطَارِ

Arabic pangram set in Amiri.

by providing the needed glyphs and shaping rules, sometimes working around the shortcomings of Unicode.

Amiri is free software, available under the terms of the SIL Open Font License (OFL) v1.1. Additional free licenses will be considered if the need arises (e.g., to remix it with another free typeface).

The development of the Amiri typeface has been supported by the TUG development fund and Google Web Fonts, as well as generous donations from enthusiastic users. Amiri also owes much of its existence to the great help offered by its users reporting bugs, testing on platforms and configurations to which I do not have access, and offering great insight on various aspects of typesetting and language support.

Short to medium-term plans for Amiri include coverage of recently added Arabic characters to Unicode, polishing the bold font, and spinning off specialized fonts, e.g., a font for Qur'an typesetting with defaults that are more suitable for Qur'an than regular text. Longer-term plans include math and display companions.

Amiri has been developed exclusively using free software, mainly FontForge, Inkscape, Python and VIM.

◇ Khaled Hosny
http://amirifont.org

Biber — the next generation backend processor for BibLaTeX

Philip L. Kime

Abstract

For many, particularly those writing in the humanities, Philipp Lehman's BibLaTeX package has been a much welcomed innovation in LaTeX bibliography preparation. The ability to avoid the BibTeX stack language and to be able to write sophisticated bibliography styles using a very rich set of LaTeX macros is a considerable advantage. Up until 2009 however, BibLaTeX still relied on BibTeX to sort the bibliography, construct labels and to create the `.bbl`. The requirement for a dedicated backend processor to do such tasks was not going to go away as doing complex, fast sorting in TeX is not a particularly amusing task. It was clear that in the future, the BibLaTeX backend processor needed to be able to handle full Unicode and many feature requests were being raised for things which the backend had to do and which were either impossible or nightmarish to do with BibTeX. Biber was created to address these issues and this article is about how it works and the many rather nice things it can do. Biber is the recommended backend processor for BibLaTeX, replacing BibTeX. There will come a time (probably around BibLaTeX 2) when BibTeX is no longer supported for use with BibLaTeX, so read on ...

1 History

François Charette originally started to write Biber in 2008 and after I realised that an APA style I was writing for BibLaTeX required some fundamental changes to the backend processor and that BibTeX wasn't going to be it (for why, see below), I had a look at the early Biber. I played with it for a while, found a small bug and submitted it. Things escalated and development entered a very rapid period where François and I knocked Biber into a releasable shape quite quickly. After a year or so, the vicissitudes of life pulled François away and I was left to my own devices with Biber gaining users rapidly, particularly in Germany, probably due to Philipp Lehman's involvement with the development as we soon realised we had to coordinate BibLaTeX and Biber releases. This continues and BibLaTeX and Biber are now so closely linked, it is fair to say that they are essentially one product. As we approach the BibLaTeX 2.0 release, the plan is to drop BibTeX support altogether as there are so many features now which are marked "Biber only" in the BibLaTeX manual. It's those features which I will describe below.

François says that the name comes from the national animal of the last country he lived in, translated into the language of the country he currently lives in. It also sounds a bit bibliographical.

2 What Biber does

Biber is used just as you would BibTeX. It's designed to be a drop-in replacement for BibLaTeX users. It uses a BibTeX compatible C library called "`btparse`" and so existing `.bib` files should work as-is. When BibLaTeX is told that it's using Biber instead of BibTeX as the backend processor, it outputs a special `.bcf` file. This is nothing more than a fancy `.aux` file in XML which describes all of the necessary options, citation keys and data sources which Biber uses to construct the `.bbl`. XML was a natural choice as the options can get quite complex (particularly for sorting). Biber reads the `.bcf` file, looks for the required data sources, reads them and looks for the citation keys also mentioned in the `.bcf`. Then it constructs a `.bbl` and writes it. Sounds simple? It's not. Biber is about 20,000 lines of mostly object-oriented Perl and some of the things it does are quite tricky.

3 Distributing Biber

Biber is written in Perl. This is an ideal language for such a task, as Perl 5.14 (which is what Biber uses now) has full Unicode 6.0 support and some really superb modules for collating UTF-8 which have CLDR[1] support, allowing sorting to be tailored automatically to the idiosyncrasies of particular languages. The `Text::BibTeX` module makes parsing BibTeX files easy but I had to change the underlying `btparse` C library a little bit to make it deal with UTF-8 when forming initials out of names and to address a few other things which are the inevitable consequences of a library written probably fifteen years ago; other than that, the library has proven to be a solid foundational element of Biber. I have to thank Alberto Manuel Brandõ Simões, the current `Text::BibTeX` maintainer for being so flexible and releasing new versions so quickly after my hacks.

Distributing Perl programs with such module dependencies is not easy and was a major stumbling block to early adoption of Biber. Then I came across the marvellous `PAR::Packer` module which allows one to package an entire Perl tree with all dependencies into one executable which is indistinguishable from a "real" executable. One virtualised build farm later and Biber had an automated build procedure for most major platforms and was swiftly put into TeX Live. Now all users have to do is to update their

[1] Common Locale Data Repository

TL installation and type "`biber`". SourceForge[2] is home to regular updates of the development binaries and github[3] is home to the Perl source which can be used instead of the binary versions if you don't mind installing some Perl modules (in fact, I only ever use the Perl source version myself).

4 Unicode and sorting

One of the main issues with the original BibTeX is that it is ASCII only. There is an 8-bit version `bibtex8` but that's not really enough these days. There is also a newish Unicode version `bibtexu` but that doesn't help BibLaTeX's myriad of other needs for its backend and it doesn't help with CLDR and the hard problem of complex sorting.

Biber is Unicode 6.0 compliant throughout, even the file names it reads and the citation keys themselves. This means that your data sources can be pure UTF-8 which is particularly nice if you are using a UTF-8 engine like XeTeX or LuaTeX. In fact, Biber will look at the locale settings passed by BibLaTeX (or those found in the environment or passed on the command line) and automatically (re)encode things to output a `.bbl` in whatever encoding you want. It will even automatically convert UTF-8 to and from LaTeX character macros/symbols in case you are using a not-quite-Unicode engine like pdfTeX.

Sorting is one of the most important things that Biber does. Sorting the bibliography is done by default using the UCA (Unicode Collation Algorithm) via the excellent `Unicode::Collate` module. This is CLDR aware and so it will take notice of the locale from various sources and tailor the sort accordingly. Swedes hate it when ä sorts before å and CLDR support avoids upsetting Swedes. Sorting a bibliography means dealing with sorting requirements such as:

> "Sort first by name (or editor if there is no name or translator if there is no editor) and then descending by year and month (or by original year and month of publication if there is no year) and then by just the last two digits of the volume and then by title (but case insensitive for title). Oh, and if there is a special shorthand for the entry, sort by that instead and ignore everything else."

Biber does this in complete generality using a multi-field sorting algorithm allowing case sensitivity, direction and substrings to be specified on a per-field basis. BibLaTeX defines many common sorting schemes (such as name/year/title, etc.) but you are free to define your own using a nice LaTeX macro interface. This interface makes BibLaTeX write a section in the XML `.bcf` which Biber reads to construct the sorting scheme it uses to sort the entries before writing the `.bbl`. I am not aware of any bibliography system that has better sorting but that may be wishful thinking born of spending so much time getting it to work ...

5 Data sources and output

It may have struck readers as strange that I refer to their `.bib` files as "data sources". This is because Biber can read more than just BibTeX format files. It has a modular data source reading/writing architecture and so new drivers can be written relatively easily to implement the ability to read new data sources and write new output formats. Data sources are read and internal entry objects constructed so that the data is processed in a source-neutral format internally. Currently, Biber can also read files in RIS format, Zotero XML/RDF format and Endnote XML format but support for these formats is experimental, partly due to weaknesses in the formats themselves, it has to be said. There is support for remote data sources for all formats by specifying a URL that returns a file in the format. This is quite useful with services such as CiteuLike which has a `.bib` gateway.

Biber normally outputs a `.bbl` file but it can also output a GraphViz `.dot` file which allows you to visualise your data. This is mainly useful for checking complex cross-reference inheritance and other entry-linking semantics. Biber can also output BibLaTeXML which is an experimental XML data format specially tuned for BibLaTeX (of course it can read this too).

A very nice feature of Biber is the "sourcemap" option. It is often the case that users would like to massage their data sources but they have no control over the actual source. Biber allows you to specify data mapping rules which are applied to the data as it is read, effectively altering the data stream which it sees, but without changing the source itself. For example, you can:

- Drop all ABSTRACT fields as the entries are read so that their strange formatting doesn't break LaTeX.
- Add or modify a KEYWORD field in all BOOK or INBOOK entries which come from a data source called "`references.bib`" whose TITLE field matches "Collected Works" so that you can split your bibliography using BibLaTeX filters.
- Use full Perl regular expressions to match/replace in any field in the entry to regularise messy variants of a name so that the same-author disambiguation features of BibLaTeX work nicely.

[2] `https://sourceforge.net/projects/biblatex-biber`
[3] `https://github.com/plk/biber`

The "sourcemap" option is quite general and provides a linear mapping interface where you can specify a chain of rules to apply to each entry as it is read from the data source. The Biber PDF manual has many examples.

6 Uniqueness

A major feature is the automated disambiguation system. Depending on the options which you set in BibLaTeX, Biber will automatically disambiguate names by using either initials or, if necessary, full names. Even better, it can, if you like, disambiguate lists of names which have been truncated using "et al." by expanding them past the "et al." to the point of minimal unambiguity. (This is a requirement for APA style and the very feature I needed when I started looking at Biber. It took two years to get this implemented.) This is fairly deep magic as it interacts with name disambiguation in an unbounded loop sort of way.

The disambiguation system can be asked to do more subtle types of work too, such as disambiguating citations just enough to make them unambiguous pointers into the bibliography but not enough to make every single individual author unambiguous, etc. These are quite fine points and make sense when you read the section of the BibLaTeX manual which covers this, with examples. Again, I don't know of any other bibliography system that has automated this.

7 Other features

The following features are all due to feature requests by BibLaTeX users and some were quite complex to implement. Some of them are waiting until BibLaTeX 2.x for a macro interface to expose them to users as this is when it is planned to retire BibTeX support from BibLaTeX.

- Many BibLaTeX options can be set on a per-entrytype basis so you can, for example, choose to truncate names lists of five or more authors with "et al." for BOOK entries and choose a different limit for ARTICLE entries.
- Biber only needs one run to do everything, including processing multiple sections.
- You can create an entry "set" (a group of entries which are referenced/cited together) dynamically, just using BibLaTeX macros. With BibTeX, this requires changes to the data source.
- "Syntactic" inheritance via a new XDATA entrytype and field. This can be thought of as a field-based generalisation of the BibTeX `@STRING` functionality (which is also supported). XDATA entries can cascade so you can inherit specific fields defining a particular publisher or journal, for example.
- "Semantic" inheritance via a generalisation of the BibTeX cross-reference mechanism using the CROSSREF field. This is highly customisable by the user — it is possible to choose which fields to inherit for which entrytypes and to inherit fields under different names etc. Nested cross-references are also supported.
- Support for related entries, to enable generic treatment of things like "translated as", "reprinted as", "reprint of" etc. (BibLaTeX 2.x)
- Customisable bibliography labels for styles which use labels (BibLaTeX 2.x)
- Multiple bibliography lists in the same section with different sorting and filtering. (BibLaTeX 2.x)
- No more restriction to a static data model of specific fields and entrytypes. (BibLaTeX 2.x)
- Structural validation of the data against the data model with a customisable validation model (BibLaTeX 2.x)

Feature requests and bug reports are always welcome via the SourceForge tracker.

◇ Philip L. Kime
Zürich, Switzerland
`Philip (at) kime dot org dot uk`
`http://biblatex-biber.sourceforge.net`

XeLaTeX and the PDF archivable format

Claudio Beccari

Abstract

At this time, XeLaTeX produces a final PDF output file but it gets this output by means of the transformation of a XDV (extended DVI) intermediate file. This excludes most of the possibilities offered by pdfLaTeX that, since version 1.40.9 and with the help of an extension file `pdfx.sty`, can directly produce a PDF/A-1b compliant output. This paper explains how to overcome this by resorting to the ubiquitous Ghostscript program.

1 Introduction

Several papers have been already published in TeX-related journals, ArsTeXnica and *TUGboat* included, about producing PDF/A-compliant archivable files. Almost all of these papers focused on the various *caveats* that are necessary to observe in order to avoid the many pitfalls of this format. Some papers also discussed the fact that the color profiles are not so clearly defined, so sometimes a non-compliant file is obtained just because an unsuitable color profile file has been employed. Some papers pointed out that sometimes the Preflight program, the most authoritative one included in the Adobe Acrobat Pro suite, fails to recognize file compliance with the ISO standard labelled PDF/A-1a or PDF/A-1b. But to the best of my knowledge, no paper has dealt with producing a PDF/A-compliant file from a source intended to be composed with XeLaTeX.

Let us recall some pieces of information. The PDF/A ISO standard was devised in 2005, and regulated with the ISO-19005-1:2005 document. This standardizes two sub-formats, labelled 1a and 1b, with the latter being less stringent than the former; it requires that the level of the PDF language used in the PDF file is level 4; it requires the fonts to be outlines and that they be subset-embedded in the document file; it requires that the color profiles are clearly defined, and it requires the presence of certain metadata, in a non-encrypted form, so that library searches can be performed. The purpose is to have files that fulfill specific limitations, but that will be readable from now on for an unlimited length of time, in spite of the fact that in, say, fifty years the fonts and the programs available today may not exist any more. The former sub-format, 1a, is more stringent in the sense that the fonts must be completely embedded and also the document structure must be included in the file.

As far as I am aware, archivable files in the 1b subformat are generally sufficient for the purpose of long-term reproduction on screen of the archived documents, exactly as the authors intended them to be. Therefore I will concentrate on this "simpler" format; besides, as a practical matter, I did not find any means for producing the 1a format except the Preflight program of Adobe Acrobat Pro, to which I have no direct access.

Finally it must be noticed that a new standard has been issued in 2011, ISO 19005-2:2011, that slightly extends the previous PDF/A standard; with this new standard, PDF language level 6 may be used and JPEG2000 images may be included in the archivable documents. Although these are important enhancements in certain areas, I will not deal with them and just stick with the previous standard, for no other reason than that the Ghostscript program, which is needed for the task, is not yet capable of satisfying the new standard. I hope that in a short time Ghostscript will be updated and its documentation will show the small modifications that need to be introduced into the necessary scripts.

2 PDF/A requirements

2.1 The metadata

Any PDF/A compliant file must contain some metadata that describe some features of the document, from the color profile and the document title and author, to the keywords that ease library searches of the archived document. Some metadata are compulsory, some are optional.

As for the compulsory metadata, I show below how to prepare a suitable auxiliary file that contains all the necessary information in the PostScript language; Ghostscript will translate such information into the XML language and will insert this XML code into the output file.

In some sense this is the simplest part of the whole procedure; the problem consists of knowing what information to supply and in which form.

2.2 The color profile

One of the most mysterious pieces of information is the one that describes the color profile; Luigi Scarso already wrote a paper [9] where he discusses this problem in connection with the typesetting program ConTeXt MkIV. But the main problem is not the particular typesetting program, but rather the very concept of *color profile*. I won't go any deeper into this question, but rather redirect the reader to [9], where the question is thoroughly discussed. Here, I

Editor's note: First published in ArsTeXnica #11, April 2011. Reprinted with permission, in slightly different form.

remark that I have obtained satisfactory results by downloading the color profile file `ECI-RGB.V1.0.icc`, freely downloaded from the `www.eci.org`. This file may be saved anywhere that Ghostscript can find it, but I suggest saving the file in a system-wide folder and to specify an absolute path when dealing with this file.

This color profile is generic, and yields satisfactory results in most circumstances; it deals only with the RGB (red, green, blue) additive color model (used by, for instance, TV and computer screens) and the images inserted into the document file should be consistent with this color model. Therefore, no image in the CMYK (cyan, magenta, yellow, black) subtractive color model should be used in a PDF/A-compliant file when the color profile refers to the RGB color model.

2.3 Hyperlinks

PDF/A-compliant files may use internal hyperlinks in order to ease the document navigation. "Internal" links means that link targets are internal to the document, so the reader may use the bookmark pane of a PDF reader to jump from one point to another in the document. This possibility is particularly useful in a reference document.

External links are prohibited in PDF/A; external links refer to other documents on the same computer, or to targets in the Internet. It is evident that a long term archiving process cannot have links between objects that may exist today, but most likely will not exist in fifty years. Therefore, when archiving is a requirement, any document should be self contained; if one needs to refer to another document, include either a complete reference in the document bibliography, or the referenced document in its entirety. The choice depends on the author, but we must keep in mind the requirement that the archived document *must* be self contained.

This is not a trivial constraint; after all, it's our daily experience, for anyone using the Internet, that many Internet addresses valid yesterday are not valid today, not to mention fifty years from now!

In order to be sure to have the hyperlinks behave as they should with PDF/A-compliant documents, it is sufficient to specify the `pdfa` option either to the class itself, or to the `hyperref` package directly, or to the `\hypersetup` command with which the package may be customized.

2.4 Fonts

The default fonts of any TEX distribution are pretty good for most purposes, but they fail in some other instances. I already wrote a short paper on this subject [2] in order to find a patch to "heal" the `cmsy*` fonts that use some zero-width glyphs. This happened with the math family three fonts, the one that contains the math symbols, because some symbols, such as $\mapsto$ and all the negated relation operators $\neq$, $\notin$, etc., resorted to the superposition of a zero-width glyph over, or beside, another symbol. Zero-width glyphs are not PDF/A compliant.

With XELATEX the font choice is much wider, although where math typesetting is concerned, at this time there are only a few suitable OpenType fonts; but, as Unicode-encoded fonts, they are (or should be) safe under the point of view of compliance with the PDF/A requirements. I confess that I did not test for this conformity the recently available OpenType Latin Modern Math fonts, but I tested the XITS math fonts (with the widest choice of math glyphs) and I did not find any glitch.

On the other hand, I found out something that either I neglected in the past or that more modern checking programs can spot: some problems with the use of accents in the "normal" Type 1 Computer Modern `OT1`-encoded fonts.

Evidently if you use XELATEX you have no problem in avoiding the traditional Computer Modern fonts; if you like their shape, you'd rather use the OpenType CM-Unicode fonts (which contain the full accented set of Latin letters, Cyrillic letters, and Greek letters, suitable for the Greek polytonic orthography), and there would not be any problem. But the problem might show up when you include in your document PDF files or pages extracted from other documents, where the latter were typeset with the default CM fonts.

I may have overlooked this problem in the past, because I never use `OT1`-encoded fonts for typesetting my documents; moreover, in principle nobody should use the `OT1`-encoded fonts if the document they typeset contains even a single accented word. The poor performance of the `OT1`-encoded fonts depends on the fact that accents are superimposed on any letter that has to be accented by an overlay mechanism that is visually acceptable when a document is printed, but in effect is a poor patch that does not follow the best practice used by the OpenType fonts and the `T1` encoded "normal" TEX fonts: OpenType fonts use accented letters and `T1` encoded fonts translate the accenting process into a glyph substitution so that no overlay process is involved.

Therefore it is most important to check the type and quality of the fonts embedded into a PDF file to be included, be it a technical diagram or a stretch of text. For this purpose I find the free program Adobe Reader X (any recent version will do the

task) extremely useful, where pressing ctrl+D (or cmd+D on Mac computers) opens a dialog pane where the user can select the "Fonts" tag and get the full list of the fonts contained in the document. In particular, the user can require that no Type 3 (bitmapped) fonts are used and, if any CM font is used, the user may examine the document to find out if any accented letters have been used. In the latter case the user should process the document to be included by following the procedure shown in the next section.

2.5 Included documents and files

If the documents are in PDF format, the Adobe Reader procedure above may be used also for checking other document characteristics. This or other programs should be used also for controlling the color model of the documents or images to be included.

The user must distinguish between bitmap and vector images. The former may contain anything, from photographs to text and line art; they must be controlled only to determine the color model. Should it be a "wrong" model, almost any image processing program can be used to open the file and save it again with a different color model. Remember the only color model usable for PDF/A-compliant images is RGB, though the "gray" model is also admissible, since the RGB model can render gray nuances very well. But while bitmap format may be acceptable for photographs with a pixel density of at least 300 dpi, in general it is not valid for line art and textual files. The compression method used by the JPEG format (`.jpg`) is not lossless, therefore the reproduced image may exhibit unwanted artifacts, often very visible if the image contains periodic patterns. The Portable Network Graphics format (`.png`) uses a lossless compression method, and in general is more suited for line art. Vector graphics, in any case, are the best suited for line art, but sometimes it's not possible to have every line art picture available in such a format.

But stretches of text and vector graphics lose all quality in a terrible way if they are converted to bitmapped form; therefore the user should pay much attention to what s/he does with files to be included that do not comply with the PDF/A standard.

Concerning `.eps` vector files, it's easy to convert them to `.pdf`, but it's necessary to pay attention that every possible font glyph is embedded into the transformed file.

One way to correct the font problems of a PDF file is to open it with the free program `inkscape` and save it back to PDF format. In this process the vector fonts are converted into vector drawings that reproduce the same glyphs, but do not exploit or exhibit any font property. In this way, for example, it is possible to draw the glyphs of the fonts that were not embedded into the file, and it is possible to redraw the poor accent overlay of the CM fonts. I confess I myself have never tried this procedure, but it was suggested by Hàn Thế Thành [4].

3 The metadata auxiliary file

We are almost ready to produce a PDF/A-compliant file from a file typeset with XeLaTeX. I assume that XeLaTeX has been run the sufficient number of times so as to be sure that the final PDF document does not require any further adjustment. Of course we can repeat the transformation, but it's better to wait until the document is substantially stable. We are going to use Ghostscript; it must be at version 8.60 or higher; the more recent the better. At writing time I am using version 9.02.

As already mentioned, we need an auxiliary file to specify the metadata and to set up some special PostScript commands necessary for this task. The Ghostscript distribution contains a file named `PDFA_def.ps` contained in the sub-tree `.../ghostscript/`⟨*version*⟩`/lib/`; where this sub-tree is grafted into your general system tree depends on your platform and your operating system, but you can execute a search command and find out where all this resides if need be.

Copy the above-mentioned file `PDFA_def.ps` to the folder where you saved your document master file. Suppose this master file is named `mydoc.tex`; copy the above `.ps` file to `mydoc-def.ps` (notice I changed the underscore into a normal "hyphen sign" or "short dash").

Now open this file with an ASCII editor; since it is a `.ps` file you might right-click the file name, but you have to choose the editor name yourself, since you cannot use the default application for `.ps` files. On a Windows platform you might use `Wordpad`; on GNU/Linux, you might use `vim`, `emacs`, `gedit`; on a Mac, you might use `Textedit.app` or `TextWrangler.app` or `TeXworks.app`, but avoid using `TeXShop.app`, since TeXShop is capable of "distilling" a `.dvi` or `.ps` file into PDF format, which generally is a very useful feature — but not in this case!

The newly created `mydoc-def.ps` contains a few lines commented with a `% Customize` comment; you should edit these lines in the way that is best suited to introduce the necessary metadata information. To insert the metadata suitable for this article, I would use the file as shown on page 19.

As you can see, there are three sets of data that can be customized:

1. The `/ICCProfile`; I changed the default to the

The auxiliary .ps file

```
%!
% $Id: PDFA_def.ps 8284 2007-10-10 17:40:38Z giles $
% This is a sample prefix file for creating a PDF/A document.
% Feel free to modify entries marked with "Customize".

% This assumes an ICC profile to reside in the file (ISO Coated sb.icc),
% unless the user modifies the corresponding line below.

systemdict /ProcessColorModel known {
...
} if

% Define entries to the document Info dictionary:
/ICCProfile (/Users/claudio/icc/ECI-RGB.V1.0.icc) def % Customize

[ /Title (XeLaTeX and the PDF archivable format) % Customize.
/Author (Claudio Beccari) % Customize.
/Subject (How to produce a PDF/A compliant document typeset with XeLaTeX) % Customize.
  /DOCINFO pdfmark

% Define an ICC profile :
[/_objdef {icc_PDFA} /type /stream /OBJ pdfmark
[{icc_PDFA} <</N systemdict /ProcessColorModel get
          /DeviceGray eq {1} {4} ifelse >> /PUT pdfmark
[{icc_PDFA} ICCProfile (r) file /PUT pdfmark

% Define the output intent dictionary :
[/_objdef {OutputIntent_PDFA} /type /dict /OBJ pdfmark
[{OutputIntent_PDFA} <<
  /Type /OutputIntent             % Must be so (the standard requires).
  /S /GTS_PDFA1                   % Must be so (the standard requires).
  /DestOutputProfile {icc_PDFA}            % Must be so (see above).
  /OutputConditionIdentifier (CGATS TR001)      % Customize
>> /PUT pdfmark
[{Catalog} <</OutputIntents [ {OutputIntent_PDFA} ]>> /PUT pdfmark
```

The shell script

```
#!/bin/bash
file1=$1.pdf
file2=$1-a.pdf
file3=$1-def.ps
# WHAT FOLLOWS MUST BE WRITTEN ON JUST ONE LINE:
gs -dPDFA -dNOOUTERSAVE -dUseCIEColor -dCompatibilityLevel=1.4 -sDEVICE=pdfwrite
   -sProcessColorModel=DeviceCMYK -sPDFACompatibilityPolicy=1 -o "$file2" "./$file3" "$file1"
```

above-mentioned file `ECI-RGB.V1.0.icc`, giving the full path from the root of my disk to the file; probably I could have abbreviated my home folder with `~`, but the full path is more easily changeable on those Windows platforms that do not have the notion of "home".

2. The document `/Title`, `/Author` and `/Subject`; other similar metadata may be added in a similar way, but remember: these metadata have nothing to do with those you can specify in the input files to be processed with `pdfLaTeX` by means of specific `\pdf...` commands; some of those commands may have a meaning also for XeLaTeX, but their contents do not migrate to the proper PDF/A file section where metadata should be.
3. The `/OutputConditionIdentifier`; I did not modify it at all.

Now the fact that we named this auxiliary file with relation to the main document comes in handy,

because it is clear that every document needs its own auxiliary file. We thus avoid the possibility of fiddling with a myriad of `PDFA_def.ps` files, all with the same name, even if they are in different folders, and with Ghostscript looking for it starting with its own sub-tree; otherwise, you may get very frustrated trying to figure out why Ghostscript does not fetch your newly-created `.ps` file.

4 The script

Ghostscript requires a long series of specifications for doing its job; shell scripts (for Unix-like systems) or bat files (for Windows) are good for specifying the whole command string without errors and without the risk of forgetting some terms.

The shell script on page 19 may be copied for use; the only attention that must be paid is to eliminate the end-of-lines in the last long command; it must consist of just one line. The shell script may be changed into a bat file by using the Windows name for the Ghostscript executable (`gswin32c` in place of `gs`), using `%1` in place of `$1`, and changing the comment character `#` to the string `REM`. The first three `file` assignments are not really necessary; one might avoid them and use only one symbolic parameter, but they might be used for testing the presence of the files that are required and to issue the necessary messages in case they were missing. The bash file might be saved with the name `pdf2pdfa` (or `pdf2pdfa.bat`) without forgetting to give it the proper mode bits, for example, `chmod 755 pdf2pdfa`.

Now the user may simply issue on the terminal the command:

```
pdf2pdfa mydoc
```

and Ghostscript will do the work.

Of course the last step is to check PDF/A compliance with a reliable program, such as Preflight. Please do not trust the information issued by Adobe Reader X when it opens a PDF file that pretends to be PDF/A-compliant. The information is a wish, in the sense that the statement might be true, but the document has not been really tested in every remote corner of its compliance. So the real message should not be: "This file is PDF/A compliant", but: "This file might be PDF/A compliant".

5 Conclusion

I have converted a number of PDF documents produced with XeLaTeX using the procedure described in the previous sections. Of course I have observed all the caveats I listed above; in fact, they are the result of my failing efforts. I had to find out at every failure the cause, but eventually I could put together a reasonable list of caveats.

There is another thing that might be done: to write an extension file to be read by the main document file, that accepts LaTeX style commands in order to specify the metadata and write the auxiliary file without any specific intervention by the user; this extension could also run Ghostscript by means of a suitable "shell escape" command at the end of the processing by XeLaTeX, as the last task of the `\end{document}` statement. Apparently there already exist the necessary hooks, but I must leave this further task for another moment.

References

[1] Adobe. XMP toolkit SDK. `http://www.adobe.com/devnet/xmp/sdk/eula.html`, 2010.

[2] Claudio Beccari. Some PDF/A tricks. *The PracTeX Journal*, 2010. `http://tug.org/pracjourn/2010-1/beccari`.

[3] Olaf Drümmer, Alexandra Oettler, and Dietrich von Seggern. *PDF/A in a Nutshell: Long Term Archiving with PDF*. 2008. `http://www.pdfa.org/download/pdfa-in-a-nutshell`.

[4] Hàn Thế Thành. Generating PDF/A compliant PDFs from pdfTeX. `http://support.river-valley.com/wiki/index.php?title=Generating_PDF/A_compliant_PDFs_from_pdftex`, 2008.

[5] Donald E. Knuth. *The TeXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1990.

[6] Emanuela Martini. Lo standard PDF/A: Sperimentazione di software per la verifica di conformità allo standard ISO 19005:2005, 2007.

[7] PDF/A. Technical notes. `http://www.pdfa.org/publications`, 2005–2010.

[8] CV Radhakrishnan and Hàn Thế Thành. Generation of PDF/X-1a and PDF/A-1b compliant PDF's with pdfTeX — `pdfx.sty`. `http://mirror.ctan.org/macros/latex/contrib/pdfx`, 2009.

[9] Luigi Scarso. Introduction to colours in ConTeXt MkIV. *TUGboat*, 31(3):203–207, 2010.

◇ Claudio Beccari
Villarbasse (TO), Italy
`claudio dot beccari (at) gmail dot com`

Avoid eqnarray!

Lars Madsen

Abstract

Whenever the `eqnarray` environment appears in a question or example of a problem on `comp.text.tex`, `tex.stackexchange.com` or other fora there is a high probability that someone will tell the poster not to use `eqnarray`. This article will provide some examples of why many of us consider `eqnarray` to be harmful and why it should not be used.

Introduction

When someone asks a question on `comp.text.tex`, `tex.stackexchange.com` or other fora about the `eqnarray` environment or shows an example using it, there will always be someone that instructs the poster to stop using `eqnarray` and use something better instead. This article provides an example-based overview of some of the reasons why many people consider `eqnarray` to be obsolete. Thus, this article can be used as a reference when a poster asks for an explanation.

The prerequisites for this article are a basic knowledge of LaTeX and knowledge of the syntax used by `eqnarray`. Experience with the environments from the `amsmath` package is a plus but not mandatory.

1 The basics

In plain LaTeX, the `eqnarray` environment is basically the only construction available for numbered multi-line equations. The `eqnarray` environment is similar to

```
\begin{array}{rcl}
  ...
\end{array}
```

with the difference being that the first and last cell in each row are automatically typeset as display style mathematics, and not as text style math as it would be in the `array` environment; also, `eqnarray` supports automatic equation numbers.

The principal `eqnarray` usage is similar to this example:

```
\begin{eqnarray}
  y &=& (x+1)^2 \\  &=& x^2+2x+1
\end{eqnarray}
```

which results in (without the box):

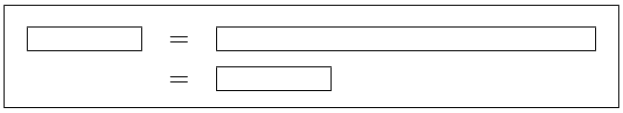
$$\begin{aligned} y &= (x+1)^2 && (1)\\ &= x^2+2x+1 && (2)\end{aligned}$$

In the examples that follow, we use the command `\dbx` instead of writing some meaningless arbitrary mathematical formula. `\dbx` is a simple macro, defined by the author, that writes a box to simulate some random mathematical material. Using an optional argument we can adjust the width of the box created by `\dbx`.

The reason for using simulated math instead of actually writing something is that removing the actual text makes the reader more aware of the actual problem, which is not the text but rather the construction/surroundings themselves. The example above will be shown like this instead:

```
\begin{eqnarray*}
  \dbx &=& \dbx[5cm] \\  &=& \dbx
\end{eqnarray*}
```

which results in:



2 Behold the problems

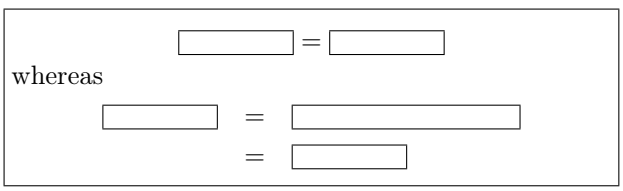
2.1 The primary problem: Spacing inconsistency

Most commonly, `eqnarray`-users write their displayed equations by mixing `eqnarray` and `eqnarray*` with `equation`, `\[...\]`, or `$$...$$` constructions. Some even mix it with environments from the `amsmath` package (though this is mostly seen when a document has been written by more than one author).

This mixing results in the primary problem with `eqnarray` — *spacing inconsistency*. In the following example we consider a single line equation versus a multi-line `eqnarray` equation.

```
\[ \dbx = \dbx \]
whereas
\begin{eqnarray*}
  \dbx &=& \dbx[3cm] \\  &=& \dbx
\end{eqnarray*}
```

which results in



Can you spot the problem?

Originally published in *The PracTEX Journal* 2006-4.

It is even more obvious when we place the same code using `eqnarray` and `equation` next to each other:

```
\begin{eqnarray} \dbx &=& \dbx[3cm]
\end{eqnarray}
  versus
\begin{equation} \dbx = \dbx[3cm]
\end{equation}
```

which results in

$$\boxed{\qquad} \;=\; \boxed{\qquad\qquad\qquad} \tag{3}$$

versus

$$\boxed{\qquad} = \boxed{\qquad\qquad\qquad} \tag{4}$$

Can you see the difference?

We notice how the spacings around the ='s are inconsistent, i.e., not equal. Consistency being one of the key values in any good document design, the spacing around the = signs should be equal on both sides (not counting stretch), no matter which construction is used.

Since `eqnarray` is (naively) built on top of the `array` environment we still have the `\arraycolsep` space between columns, which then affects the spacing around the ='s in our case. We could change the value of `\arraycolsep`:

```
\setlength\arraycolsep{1.4pt}% some length
\[ \dbx = \dbx \]
\begin{eqnarray*}
  \dbx & = & \dbx  \\  &= & \dbx
\end{eqnarray*}
```

Resulting in:

$$\begin{aligned} \boxed{\qquad} &= \boxed{\qquad} \\ \boxed{\qquad} &= \boxed{\qquad} \\ &= \boxed{\qquad} \end{aligned}$$

Changing the value of `\arraycolsep`, however, will also change the appearance of any other construction that might be using `array`, so this does not suffice; see the following example.

```
Before the change:
\begin{eqnarray*}
  A &=& \left(\begin{array}{cc}\dbx&\dbx\\
              \dbx&\dbx\end{array}\right)
\end{eqnarray*}
after the change:
\setlength\arraycolsep{1.4pt}% some length
\begin{eqnarray*}
  A &=& \left(\begin{array}{cc}\dbx&\dbx\\
              \dbx&\dbx\end{array}\right)
\end{eqnarray*}
```

Resulting in:

Before the change:

$$A \;=\; \left(\begin{array}{cc}\boxed{\qquad}&\boxed{\qquad}\\ \boxed{\qquad}&\boxed{\qquad}\end{array}\right)$$

after the change:

$$A = \left(\begin{array}{cc}\boxed{\qquad}&\boxed{\qquad}\\ \boxed{\qquad}&\boxed{\qquad}\end{array}\right)$$

Some people argue that this larger spacing is a good thing, that it helps understanding the equations in question. For that to be true the author should do this with every single equation, whether the equation was written using `eqnarray` or not. Consistency above all. We can plainly see that `eqnarray` does not follow the spacing conventions Knuth set out in TEX, whereas both `equation` and `\[...\]` do.

Here is another example from a set of notes I have been editing (actual code from the original unedited notes).

```
\begin{eqnarray*}
 {\cal C}_{0} &\subseteq& {\cal C}\subseteq
 \sigma ({\cal  C}_{0},{\cal N}) ,
\end{eqnarray*}
```

$$\mathcal{C}_0 \;\subseteq\; \mathcal{C} \subseteq \sigma(\mathcal{C}_0, \mathcal{N}),$$

Which makes one wonder if LATEX authors even notice the difference in spacing, or do they just accept it as a fact of life?

Even though `eqnarray` might not be recommended for one-liners, they do still appear quite a lot in the 'wild'.

As `eqnarray` is the only multi-line construction for plain LATEX, what should be used instead? Short answer: Use the environments from the `amsmath` package, in particular the `align` environment.

Longer answer: There are a few packages that can help including `nath`, `mathenv` and `amsmath`. Using `amsmath` is highly recommended since it is already included as part of every LATEX installation.

For those not familiar with the `amsmath` package we present a few useful constructions in Appendix A.

2.2 Problem #2: eqnarray can overwrite equation numbers

Given a long formula which happens to fit within one line were it not for the equation number, `eqnarray` will happily just ignore the equation number, without any warnings.

```
\begin{eqnarray}
  \dbx &=& \dbx[5cm]
\end{eqnarray}
```

= (5)

It can get even worse. Assume we are using the `leqno` class option, i.e. equation numbers on the left. Then assume we have a math line that is slightly longer than the text width:[1]

```
Left text edge \hfill right text edge%
\begin{eqnarray}
  \dbx &=& \dbx[5.7cm]
\end{eqnarray}
```

Left text edge right text edge

(6) =

No offence, but why on earth is `eqnarray` moving the equation number? Let us see what happens if we take the same example and switch back to equation numbers on the right:

```
Left text edge \hfill right text edge%
\begin{eqnarray}
  \dbx &=& \dbx[5.7cm]
\end{eqnarray}
```

Left text edge right text edge

= (7)

Sigh...

Well, at least that will teach authors to remember to break their equations properly.

At least the environments from the `amsmath` bundle take the equation number into consideration. Here is an example using `align`:

```
\begin{align}
  \dbx &= \dbx[5.7cm]
\end{align}
```

= (8)

2.3 Problem #3: Silence of the labels

Part of my job is to process a preprint series published by my department. This brings me into contact with many different styles of LaTeX writing and usage. One thing that I frequently do (as part of my visual improvement procedures) is convert `eqnarray` environments into `align` environments (or similar). This is where one starts to find the hidden label errors. Most often these occur when two or more people have been writing/editing the same file.

Here is the first example:

[1] Example provided by Barbara Beeton.

```
\begin{eqnarray}
  \dbx & = & \dbx \\
  \dbx & = & \dbx \label{eq:2} \nonumber
\end{eqnarray}
From equation (\ref{eq:2}) we conclude
\begin{equation}
  \dbx=42.
\end{equation}
```

So the author had an equation which he or she no longer wanted to have numbered (`\nonumber`). Which is perfectly reasonable, but the author neglected to check whether the now-dead label (`eq:2`) was referred to. The result is as follows:

= (9)

=

From equation (10) we conclude

= 42. (10)

Huh? This might end up as an interesting form of argumentation. It seems as if `eqnarray` actually steps up the equation counter at the start of every line (hence `\label` catches something) and when it encounters `\nonumber` it does not write any equation number and steps the equation counter one down again. On a side note, `equation` has the same problem if one mixes it with `\nonumber` (something which is *not* fixed by using `amsmath`).

The worst thing here is that `eqnarray` does this silently, without warnings, so if you do not know that this might happen you will never notice it unless someone carefully reads the article.

As it happens, I recently received an article which showed exactly the same problem in `eqnarray*`. Here one only has to place a label inside a non-numbering `eqnarray*` (we use `\theequation` to show the current value of the equation number):

```
Current equation number: \theequation
\begin{eqnarray*}
  \label{eq:4}
  \theequation & = & \dbx
\end{eqnarray*}
The reference is (\ref{eq:4}).
Current equation number: \theequation
```

Resulting in:

Current equation number: 10

11 =

The reference is (11). Current equation number: 10

Who smells a rat? So, even in `eqnarray*` the equation counter is stepped up, and later stepped down at the end of each line. As we have seen, this is a problematic approach.

2.4 Problem #4: The amsthm package vs. the eqnarray environment

If one uses the `amsthm` package, and its `proof` environment, then you will get automatic placement of an "*end of proof*" marker. Sometimes one ends a proof with a displayed formula and may want to place the end marker near the equation number. This may be achieved by simply issuing `\qedhere` on the last line of the formula.

```
\begin{proof}
  \dots
  \begin{equation*}
    a=0. \qedhere
  \end{equation*}
\end{proof}
```

Proof. ...

$$a = 0. \qquad \square$$

This handy little feature, as one might guess by now, does *not* work with `eqnarray`!

3 Solution

The best solution is to *not* use the `eqnarray` environment at all. Use the environments from `amsmath` instead. If in some case that will not do, the `mathenv` package reimplements `eqnarray` to work more rationally. It also removes the restraint on the number of columns in an `eqnarray`. (Unfortunately, `mathenv` is not compatible with certain useful modern packages, notably `siunitx`.)

Sadly we see many journals and publishers who still recommend (or at least mention) the use of `eqnarray` in their guides for authors.

A The amsmath package

For more information about `amsmath` see [2], [1] and [4] (in order of recommended reading). This appendix gives a few interesting constructions, mainly showing replacements for common `eqnarray` usage.

All of the following examples require `amsmath`, hence the document preamble must include:

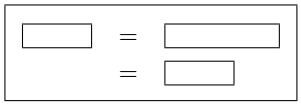
```
\usepackage{amsmath}
```

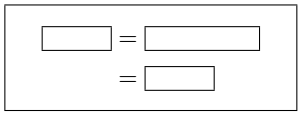
One thing to note about `amsmath` is that *every* environment from `amsmath` that provides equation numbers also has a `*`-version which does not. The package also includes an `equation*` environment which is missing from plain LaTeX.

Now the first thing we need is a replacement for `eqnarray`. We choose `align`, which has a slightly different syntax than `eqnarray`:

```
\begin{eqnarray*}
\dbx &=& \dbx[1.5cm]\\
     &=& \dbx
\end{eqnarray*}
```

```
\begin{align*}
\dbx &= \dbx[1.5cm]\\
     &= \dbx
\end{align*}
```



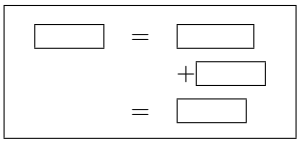


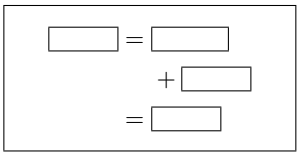
Note the reduced number of `&`'s.

Here is another common `eqnarray` construction and its `align` counterpart:

```
\begin{eqnarray*}
\dbx &=& \dbx[1cm]\\
     & & + \dbx   \\
     &=& \dbx
\end{eqnarray*}
```

```
\begin{align*}
\dbx = {} & \dbx[1cm]\\
          & + \dbx   \\
     = {} & \dbx
\end{align*}
```

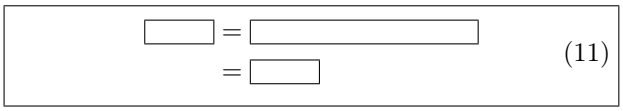




Notice the use of `{}` when the `&` is placed to the *right* of a relational symbol. Also note that the spacing around the `+` is correct in the `align` case but not when using `eqnarray`.

One construction not easily achieved with base LaTeX is a formula spread over several lines but with only one equation number for the entire formula. Again, this is easy using constructions from the `amsmath` package:

```
\begin{equation}
  \begin{split}
    \dbx & =\dbx[3cm] \\
         & =\dbx
  \end{split}
\end{equation}
```



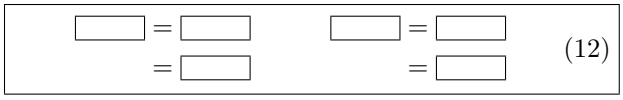
(11)

Notice how the equation number is vertically centred. The syntax for `split` is otherwise more or less the same as for `align*`.

`amsmath` also provides the `aligned` environment, which is basically the full `align` environment, but for *inner* use. (Like `eqnarray`, `split` can only have one so-called alignment column, while `align` and `aligned` can have several.)

```
\begin{equation}
  \begin{aligned}
```

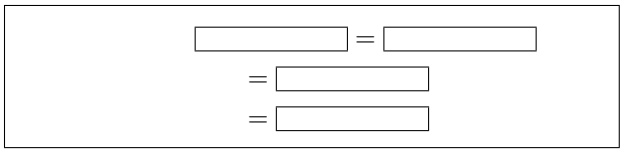
```
    \dbx & =\dbx &\qquad  \dbx & =\dbx \\
         & =\dbx &              & =\dbx
  \end{aligned}
\end{equation}
```



A.1 What about \lefteqn?

amsmath has no direct equivalent to \lefteqn, but the idea is still useful. To recap, using the \lefteqn macro inside eqnarray, one can force that particular line to be moved to the left:

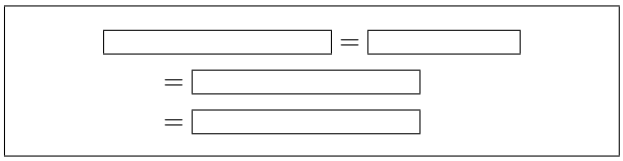
```
\begin{eqnarray*}
  \lefteqn{\dbx[2cm] = \dbx[2cm]} \\
  && = \dbx[2cm] \\
  && = \dbx[2cm]
\end{eqnarray*}
```



One usually uses this to mark the first line, and then give the impression of the rest of the lines being indented.

The mathtools package does provide an alternative, namely \MoveEqLeft:

```
\begin{align*}
  \MoveEqLeft \dbx[3cm] = \dbx[2cm] \\
       & = \dbx[3cm] \\
       & = \dbx[3cm]
\end{align*}
```



The idea is that the \MoveEqLeft marks an alignment point (which is what the ampersands follow), and then pulls the line backwards in a suitable fashion. It does *not* take any required arguments, unlike \lefteqn.

Acknowledgements

Special thanks to Barbara Beeton from the AMS for comments and suggestions for this revised version. Also many thanks to the various people who provided examples for the original version of the article.

References

[1] American Mathematical Society, *User's Guide for the* amsmath *Package*, 2002. Normally included in every LaTeX installation as amsldoc; also available via http://mirror.ctan.org/macros/latex/required/amslatex/math.

[2] Michael Downes, *Short Math Guide*, 2002. Short introduction to the amsmath and amssymb packages. ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf

[3] Morten Høgholm, Lars Madsen, Will Robertson and Joseph Wright (maintainers), *The mathtools package*, 2011. Various extensions to amsmath and others. http://mirror.ctan.org/macros/latex/contrib/mh.

[4] Herbert Voß, *Math mode*, 2006. Extensive summary describing various mathematical constructions, both with and without the amsmath package. http://mirror.ctan.org/info/math/voss/mathmode/Mathmode.pdf.

◇ Lars Madsen
Department of Mathematics
Aarhus University
Denmark
daleif (at) imf dot au dot dk

The unknown *picture* environment

Claudio Beccari

Abstract

The old *picture* environment, introduced by Leslie Lamport into the LaTeX kernel almost 20 years ago, appears to be neglected in favor of more modern and powerful LaTeX packages that eliminate all drawbacks of the original environment. Nevertheless it is still being used behind the scenes by a number of other packages. Lamport announced an extension in 1994 that should have removed all the limitations of the original environment; in 2003 the first version of this extension appeared; in 2004 the first stable version was released; in 2009 it was actually expanded with new functionality. Nowadays the *picture* environment can perform like most simple drawing programs, but it has special features that make it invaluable.

1 Introduction

Plain TeX, as described in *The TeXbook* [5], contained a simple way to draw simple graphics with `tex`. When LaTeX was first published in 1984, it contained an environment suitable for relatively complex graphics; Lamport's handbook [6] described its workings and commands. But all this seems to have fallen into complete oblivion.

Many users of LaTeX related forums keep asking questions such as "How can I produce such and such a symbol"; I keep answering "Use the *picture* environment". Apparently nobody follows my suggestions, which of course they are free to avoid; but they would save time if they spent no more than 15 minutes in reading the environment description in Lamport's handbook [6]. The second edition of this handbook announces the extensions and discusses the eliminated drawbacks; these extensions were eventually realized only in 2003 by Gäßlein and Niepraschk [4]. In 2004 the same authors released a stable version. In 2009, with the contribution of the third author Tkadlec, they released an enhanced version that added quite a few new commands that substantially extend the *picture* environment functionality. But, except for those very latest additions in 2009, everything else was already documented by Lamport in his second edition.

Not longer than a few days ago a forum participant asked how he could draw a square wave and a saw tooth wave of suitable size for setting them in line with his text: ⎍ and ⩘.

Editor's note: First published in ArsTeXnica #11, April 2011. Reprinted with permission, in slightly different form.

Here is the whole thing required to produce the two simple commands, using the recent `pict2e` extension package's new commands:

```
\newcommand*\sqwave[1][0.125ex]{{%
\unitlength=#1\relax
\picture(20,10)
\polyline(0,0)(6.5,0)(6.5,15)(13,15)%
        (13,0)(19.5,0)
\endpicture}}

\newcommand*\sawtooth[1][0.125ex]{{%
\unitlength=#1\relax
\picture(20,10)
\polyline(0,0)(6.5,0)(6.5,15)(13,0)%
        (19.5,0)
\endpicture}}
```

Most of the time suggestions are given by and to the forum participants to use external drawing programs, or to use the sophisticated PSTricks [12] or Ti*k*Z [11] packages. The former solution is generally to be avoided, because if some lettering is needed, it requires extra work to use the same fonts as those used in the document. The latter two packages are certainly capable of doing marvelous and complicated drawings, but require considerable time with the documentation and a steep learning curve.

Also, the *picture* environment has a unique feature: it can produce drawings of zero width and/or height. This special feature makes them valuable for packages such as `eso-pic` [9], `crop` [3], `layout` [8], `layouts` [15], and others. These packages draw things on the page that do not require any external package, and therefore don't have any dependency. These drawings occupy no space, although they have a specific position on the page; their contents reach whatever point on the page, as background images or marks that do not interfere with the positioning of other page elements.

In particular `eso-pic` (and similar packages for setting watermarks) and `crop` exploit this functionality specifically for setting a background picture or the crop marks in the correct positions without interfering with the other elements of the page.

At the same time within the *picture* environment it is possible to use cubic Bézier curves and to draw polygons, arches and sectors, oval frames whose corner curvature can be freely specified, white or filled circles of any dimension. The `\polyline` macro used in the above example makes it very easy to draw polylines with any number of corners and any segment slope, to the point that if the nodes are sufficiently close, it is possible to draw "smooth" curves whose points may be calculated with any number-crunching personal or mainframe computer.

2 In detail

Everybody can agree that the original *picture* environment, created by Lamport with the very first version of LaTeX, was pretty rudimentary, but there was practically nothing else to use in its place. The straight lines could have slopes that were ratios of relatively prime integer numbers not exceeding 6 in absolute value. For vectors the limitation was even stricter; the slopes could not be specified with integer numbers larger than 4 in absolute value. Why were there such strange limitations? Because straight lines were made up through the juxtaposition of small segments 10 pt (≈ 3 mm) long taken from a special font; this same special font contained also the vector arrow tips that occupied a large part of the available positions; and, remember, at that time the typesetting engine could deal only with 128-glyph fonts, so that the available short segments and arrow tips, plus a selection of closed and filled circles and/or quarter circles placed strict limitations on the drawing performance.

Patient programmers created extension packages such as `curves` [7], that could overcome such limitations by drawing lines of any slope and circles of any diameter by juxtaposing an "infinity" of small dots. For plain TeX there existed another package, PiCTeX [14], that with a suitable interface could work also with LaTeX. It performed well on large mainframes with large memory capabilities, but worked very poorly on the desktops of that age, the eighties, when a 20 MiB hard disk was a luxury and 640 KiB RAM was almost the maximum available. These packages mostly saturated the RAM and drawing was virtually impossible on personal computers.

Some progress was made when the PostScript format became available; some drawing packages (again `curves`) exploited TeX's `\special` to write raw PostScript drawing commands in the output, so that the actual action of drawing was demanded of the screen or printer driver. Nevertheless powerful extensions in this directions, such as PSTricks, appeared much later. Drawing with external programs and importing the resulting artwork was therefore a necessity, but not an easy task.

Things evolved in the right direction when personal computers, having become the universal complement of any person needing to write anything, started having a more user friendly interface, more RAM, larger hard disks, and better programs, the TeX system included. The nineties, besides the important passage from TeX 2.x to TeX 3.0, gave us LaTeX 2ε, PostScript fonts, and the drawing instrument METAPOST. This program used more or less the same philosophy that led Knuth to develop METAFONT, in order to draw the TeX system default fonts; METAPOST produced a simplified output PostScript code that was understood also by the newborn typesetting program `pdftex`. METAPOST was, and still is, fully compatible with the rest of the TeX system typesetting engines, so that all the TeX and LaTeX features could be used for putting any lettering on the METAPOST output files.

Meanwhile LaTeX went on with its small drawing environment, without exploiting the new possibilities with the PostScript format and the PDF portable document format, until Gäßlein and Niepraschk wrote the *picture* extension announced by Lamport some ten years before.

2.1 2009 extensions to *picture*

Let us now discuss the enhancements introduced by the extension realized by Gäßlein and Niepraschk. Since these changes are so recent, some commands are not described in [6].

1. First of all, the enhancements rely on the drivers that are being used to display or to print the document. More precisely, when the `latex` program is used, the `\special` commands to the driver contain only PostScript language commands; this implies a transformation of the resulting DVI file into a PostScript one by means of `dvips`, and possibly a second transformation to the PDF format by means of (for example) `ps2pdf`. On the other hand, if the document is processed with `pdflatex`, then the `\special` commands contain only the PDF language commands. Therefore the extension is fully compatible with the typical output formats provided by the most popular typesetting engines, and this is fully automatic so users need not bother about the details.
2. The output file size very often is smaller since the actual drawing computations are performed by the suitable drivers.
3. One of the limitations of the original environment was the slope of lines and vectors. In the first implementation of the extension the slope coefficients had to be integers not higher than 1000 in absolute value, thus implementing Lamport's description of 1994. The 2009 enhancements, however, remove even this limitation, and the slope coefficients can be any fractional decimal number (well, yes, not too large, not higher than 16 383.999 98 which corresponds to the the largest dimension in points that any TeX system typesetting engine can

handle). Now line and vector slopes should not have any detectable limitation.

4. The above is valid also for vectors; even better, now it's possible to pass an option to the package so that it can draw the arrow tips in "LaTeX style" or in "PostScript style". In LaTeX style the joining sides to the arrow tip are slightly concave, and the arrow shaft is straight; in PostScript style they form a polygon that resembles a stealth aircraft.
5. Circles and quarter circles were available in a limited set; now they can be drawn in any size, both filled and unfilled.
6. Line thicknesses could previously be specified only as `\thinlines` (default) and `\thicklines` (twice as thick as `\thinlines`), and only vertical and horizontal lines used the thickness specified with `\linethickness` ⟨*dimension*⟩. Now `\linethickness` can modify the thickness of all sorts of lines, Bézier splines included.
7. "Ovals", frames with rounded corners, could have the corner quarter circle with an automatic setting of its radius, in any case not larger than about 15 pt (about 5 mm), and they could not use a radius dimension specified by the document. Of course this radius should not exceed the half length of the shorter frame sides (that is, half of the distance of the longer straight lines that form the longer sides of the frame) but the radius can now be specified as an optional argument to the `\oval` command so that the created frame can have a very different look when a smaller radius is chosen compared to the same-sized frame with a larger corner radius.
8. Quadratic and cubic Bézier splines are now generated with the driver commands and they result in smooth curves, not lines with a rough contour due to the juxtaposition of many small dots. The possibility of specifying the number of dots is available even now, but it is mostly for backwards compatibility — although, even now, dotted splines might be used for special purposes. In any case they do not suffer any magnification when seen on the screen; they are scalable vector strokes. The previous command `\bezier` is maintained with its compulsory specification of the number of points to use, but two new commands, with an optional specification of the number of points, are introduced, `\qbezier` for tracing quadratic Bézier splines, and `\cbezier` for tracing cubic Bézier splines; this last command was not described in [6], and is a completely new command to the package.
9. Up to this point the traditional commands have been discussed and the differences with the original environment described. The last extension of `pict2e`, published in the second half of 2009, adds some other commands that draw other lines but in general don't require the use of `\put` to place these lines in a special position. Of course they may be shifted with `\put`, which might come in handy when fine-tuning the picture, but the `\put` is not necessary.
10. A first exception to the above statement is the new macro `\arc` that is a generalization of `\circle` (both starred and non-starred forms; in both cases the starred form produces a filled contour) which requires putting the arc center in a specific position, so that the whole command must be set as an argument to `\put`. The `\circle` command is used like this:

    `\put(`⟨*x*⟩`,`⟨*y*⟩`){\circle`⟨***⟩`{`⟨*diameter*⟩`}}`

    and similarly with the `\arc` command:

    `\put(`⟨*x*⟩`,`⟨*y*⟩`){\arc`⟨***⟩`[`⟨*ang1*⟩`,`⟨*ang2*⟩`]{`⟨*radius*⟩`}}`

    The arc has its center at point `(`⟨*x*⟩`,`⟨*y*⟩`)`, and it will go from the angle ⟨*ang1*⟩ to the angle ⟨*ang2*⟩; angles are in sexagesimal degrees and are positive in the anticlockwise direction; if the optional angles are not specified, the full circle is drawn. The arc is drawn from the smaller angle to the larger one, so that the order in which ⟨*ang1*⟩ and ⟨*ang2*⟩ is not important.
11. The following commands do not require `\put`:

    `\Line(`⟨*x1*⟩`,`⟨*y1*⟩`)(`⟨*x2*⟩`,`⟨*y2*⟩`)`
    `\polyline(`⟨*x1*⟩`,`⟨*y1*⟩`)(`⟨*x2*⟩`,`⟨*y2*⟩`)...(`⟨*xN*⟩`,`⟨*yN*⟩`)`
    `\polygon(`⟨*x1*⟩`,`⟨*y1*⟩`)(`⟨*x2*⟩`,`⟨*y2*⟩`)...(`⟨*xN*⟩`,`⟨*yN*⟩`)`
    `\polygon*(`⟨*x1*⟩`,`⟨*y1*⟩`)(`⟨*x2*⟩`,`⟨*y2*⟩`)...(`⟨*xN*⟩`,`⟨*yN*⟩`)`

    The first command is simply the segment that joins the two points identified by the two pairs of coordinates. The second command is a sequence of segments that join with one another in the order of the N specified pairs of coordinates; we have seen it at work in the example shown in the introduction. The third command is a closed polygon whose vertices are sequentially shown by the N pairs of coordinates. The fourth command is similar but draws a filled polygon.
12. In order to draw the various lines and curves, the internal commands make use of the "turtle graphics" commands used within both the PostScript and PDF languages. These elementary commands are available to the user also through package `pict2e`; they are:

    `\moveto(`⟨*x*⟩`,`⟨*y*⟩`)`
    `\lineto(`⟨*x*⟩`,`⟨*y*⟩`)`
    `\curveto(`⟨*x2*⟩`,`⟨*y2*⟩`)(`⟨*x3*⟩`,`⟨*y3*⟩`)(`⟨*x4*⟩`,`⟨*y4*⟩`)`

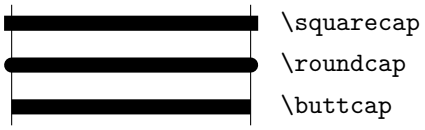


Figure 1: Ending styles for line segments (of equal width)

Figure 2: Join styles for line segments

and a few more that the reader may find in the documentation [4]. These commands may be used in any order, except `\moveto` that must fix the first position of the drawing pen. In order to finish the path it is optional to use `\closepath` in order to draw a line from the last point to the initial one, but then it is necessary to use either `\strokepath` to draw the path or `\fillpath` in order to fill the path with the default color.

13. The initial and final points of an open path may be controlled with the commands `\buttcap` (cut the path at the end points), or `\roundcap` (adjust the end points with a filled semicircle), or `\squarecap` (adjust the end points with a filled half square); in general with line art the `\roundcap` should be preferable, but sometimes it's better to use one of the other two kinds of end point finishing. See figure 1.
14. Similarly the joins between adjacent segments of a polyline or a polygon may be adjusted with the three commands `\miterjoin`,[1] `\roundjoin`, and `\beveljoin`, as shown in figure 2.

3 Examples

There are dozens of examples in the GUIT documentation [2], where every line art picture has a small legend containing the author name and the program used for producing it. This book is a collective effort of the Italian TEX users group, and is downloadable from the GUIT site `http://www.guitex.org/home/images/doc/guidaguit.pdf`. There, the interested reader can find plenty of ideas and useful "tricks".

[1] In the documentation, [4], this command is erroneously spelled `\mitterjoin`.

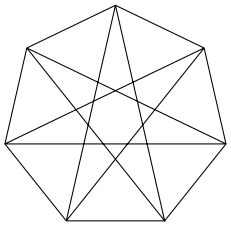


Figure 3: A heptagon with seven vertices and inscribed star

Here we present a few examples, sometimes with their source code, in order to see the modern *picture* environment at work.

A heptagon We compute the vertices of a heptagon inscribed into a circle with a diameter of 6 cm by means of a pocket calculator:

$v_1 = (1.3017, -2.7029)$ $v_5 = (-2.3455, 1.8705)$
$v_2 = (2.9248, -0.6676)$ $v_6 = (-2.9248, -0.6676)$
$v_3 = (2.3455, 1.8705)$ $v_7 = (-1.3017, -2.7029)$
$v_4 = (0, 3)$

Then we set up the picture environment (within a *figure* environment, so we don't need to do anything to limit the scope of the `\unitlength` assignment) with the following code:

```
\unitlength=5mm
\begin{picture}(6,6)(-3,-3)
\polygon(1.3017,-2.7029)(2.9248,-0.6676)%
    (2.3455,1.8705)(0,3)(-2.3455,1.8705)%
    (-2.9248,-0.6676)(-1.3017,-2.7029)
\polyline(1.3017,-2.7029)(0,3)%
    (-1.3017,-2.7029)(2.3455,1.8705)%
    (-2.9248,-0.6676)(2.9248,-0.6676)%
    (-2.3455,1.8705)(1.3017,-2.7029)
\end{picture}
```

Figure 3 contains also the seven pointed star inscribed in the heptagon.

Splines We draw some splines inside a square with sides 6 cm long; a quadratic spline has its two nodes at the square base vertices, and the control node at the center of the upper side. A cubic spline uses the four square vertices as end and control nodes:

```
\unitlength=6.5mm
\begin{picture}(6,6)(-3,-3)
\put(-3,-3){\framebox(6,6){}}
\polyline(-3,-3)(0,3)(3,-3)
\polyline(-3,3)(3,3)(-3,-3)(3,-3)
\linethickness{1.5pt}
\qbezier(-3,-3)(0,3)(3,-3)
\cbezier(-3,3)(3,3)(-3,-3)(3,-3)
\end{picture}
```

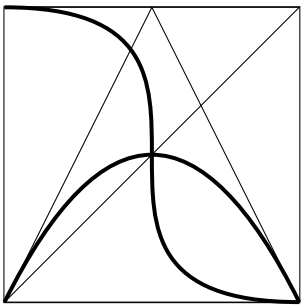


Figure 4: Quadratic and cubic splines

Figure 4 displays the result; observe the effect of the `\linethickness` assignment on the splines themselves. If you can read this document on the screen, you can magnify the picture and check the vector nature of the splines. Figure 4 contains also the polylines that join the nodes and control points, so that it's easier to see the effect of these "control segments".

An electric circuit Many years ago, at the end of the 1980s, when I had available only the *picture* environment, I needed to draw circuit diagrams. In fact, I had so many circuit diagrams to insert in my book that I needed to create suitable macros for drawing the circuit components and their connections to the various circuit nodes; of course every component had to be identified with a symbol and optionally should be assigned a value with the proper units.

Nowadays there are modular packages that work with Ti*k*Z (`circuitikz` [10]) and PSTricks (`pst-circ` [13]), but at that time there was nothing, or at least nothing I was aware of.

In my department there was a very good expert of technical drawing, and for my previous books I had asked him to draw my circuits; these drawings had to be glued to the camera ready copy, because at that time it was very difficult to insert graphical files into a document; not impossible, but difficult. The publisher, in any case, did not want any kind of file; he wanted only the camera ready copy. This procedure was pretty lengthy: draw my circuits by hand, pass them to the technician with suitable descriptions about dimensions, lettering, line thicknesses, and the like; after the drawings were done, careful checking of the correctness, the proper position of the labels and indices, and any possible typos; start again with the second draft, and so on.

Therefore I decided to write a personal package containing all the circuit macros, to work as an interface between the user and the *picture* environment with its internal macros. It took about two weeks; afterwards, I had an almost complete circuit-drawing TEX program. At that time, of course, arbitrary sloped lines were done by juxtaposing a multitude of little dots, as well as quarter, half and full circles of any diameter. Single-port components were automatically drawn as vertical or horizontal elements; connections automatically made the necessary bends in order to reach the destination nodes; two-port and four-pole devices were set in the proper orientation in order to avoid crossing their connections; operational amplifiers, nullators, norators and nullors were correctly designed; block diagrams with their signal flows, their branching nodes, their summing points, and so on, were at hand. The unit length was parametrized to the current font 'ex' unit, so that the circuit diagram would scale together with the size of the surrounding text font.

I saved much time using my macros and the technical expert eventually congratulated me, admitting that my drawings were more consistent than his own.

When the important `pict2e` package became available in 2003, I started to eliminate all references to the old tiny-point-overlay technique, and promptly switched to the new technology.

I eventually added logical components as well, so that this private package is almost complete. What is provided by `circuitikz` is much more complete, and this is the main reason why my package remains private. I keep using it for no other reason than compatibility with the past book files I wrote long ago, from which I often pick up some parts in order to assemble short tutorials for students who ask me for explanations.

The package is too large to publish here; therefore, I will not show how the user commands are realized with the internal modern *picture* environment ones. I show just the user-level code for drawing the circuit diagram of a band elimination filter:

```
\begin{circuito}(75,35)
\hconnect(0,0)(19,0)\HPolo(20,0)(65,0)
\polo(20,25)[30,25]\polo(65,25)[75,25]
\hconnect(66,0)(75,0)
\R(75,0)(75,25){R\ped{L}}
\E(0,0)(0,25){E}
\R(0,25)(19,25){R\ped{G}}
\serie*(30,0)(21,25)C{C_1}-L{L_1}
\parallelo(30,25)(55,25)L{L_2}-C{C_2}
\serie(55,0)(64,25)C{C_3}-L{L_3}
\nodi(55,0)(55,25)(30,25)(30,0)
\end{circuito}
```

and you can see the result in figure 5. As you can see the component and connection macros are very user friendly and the total amount of code for drawing

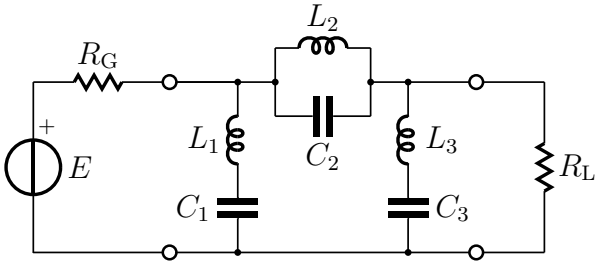


Figure 5: A band elimination filter

a complicated circuit diagram[2] is very limited. If you are reading this file on screen you can magnify the image of figure 5 and again verify that the whole drawing is made of scalable vectors. You can also recognise that the resistors are drawn by means of the `\polyline` command with the `\miterjoin` specification for the connection of the various segments.

A Cartesian diagram While teaching the synthesis of electrical circuits I often needed Cartesian diagrams of their performance; in figure 6 the squared magnitude of a fifth order elliptical filter characteristic function is plotted. The name "elliptical" derives from the use of elliptical integrals and functions of the first and second kinds. The diagram is only qualitative; although it would not have been a problem to compute the actual points by means of a suitable program, for a qualitative diagram the extreme points and the peaks and zeros should be sufficient. The whole diagram had to be also shown as a slide, therefore a `beamer` presentation was made containing the same code:

```
\unitlength=0.9mm
\begin{picture}(80,60)(-40,-5)
  \VECTOR(-40,0)(40,0)
  \Zbox(40,-2)[tr]{\omega}
  \VECTOR(0,-1)(0,55)
  \Zbox(-1,55)[tr]{|F|^2}
  \multiput(-35,5)(4,0){18}%
     {\line(1,0){2}}
  \Zbox(2,7)[bl]{1}
  \multiput(-35,45)(4,0){18}%
     {\line(1,0){2}}
  \Zbox(1,46)[bl]{H^2}
  \multiput(-2,15)(4,0){4}%
     {\line(1,0){2}}
  \Zbox(1,16)[bl]{H}
  \LINE(-10,0)(-10,-1)\Zbox(0,-2)[t]{0}
  \Zbox(-10,-2)[t]{-1}
  \LINE(10,0)(10,-1)\Zbox(10,-2)[t]{1}
```

[2] Actually the circuit diagram is not complicated at all; the complication is hidden behind the user macros, especially those for inductors, where the various Bézier cubic splines are properly described and connected to one another.

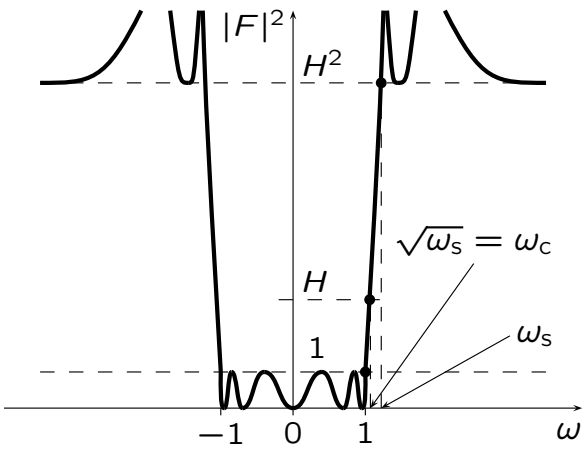


Figure 6: The squared magnitude of the characteristic function of an elliptical filter

```
{\linethickness{1.5pt}%
  \cbezier(-12.5,55)(-12,40)(-12,40)%
          (-10,5)
  \cbezier(-10,5)(-10.1,0)(-9.75,0)%
          (-9.5,0)
  \cbezier(-9.5,0)(-9,0)(-9,5)(-8.5,5)
  \cbezier(-8.5,5)(-7.75,5)(-7.75,0)%
          (-7,0)
  \cbezier(-7,0)(-5.5,0)(-5.5,5)(-4,5)
  \cbezier(-4,5)(-2,5)(-2,0)(0,0)
  \cbezier(-13,55)(-13.75,45)(-13.75,45)%
          (-14.5,45)
  \cbezier(-14.5,45)(-15.75,45)%
          (-15.75,45)(-17,55)
  \cbezier(-21,55)(-26,45)(-28,45)%
          (-35,45)
%
  \cbezier(12.5,55)(12,40)(12,40)(10,5)
  \cbezier(10,5)(10.1,0)(9.75,0)(9.5,0)
  \cbezier(9.5,0)(9,0)(9,5)(8.5,5)
  \cbezier(8.5,5)(7.75,5)(7.75,0)(7,0)
  \cbezier(7,0)(5.5,0)(5.5,5)(4,5)
  \cbezier(4,5)(2,5)(2,0)(0,0)
  \cbezier(13,55)(13.75,45)(13.75,45)%
          (14.5,45)
  \cbezier(14.5,45)(15.75,45)(15.75,45)%
          (17,55)
  \cbezier(21,55)(26,45)(28,45)(35,45)
}%
  \put(10.6,15){\circle*{1.5}}
  \put(12.2,45){\circle*{1.5}}
  \put(10,5){\circle*{1.5}}
  \multiput(12.2,0)(0,4){11}%
     {\line(0,1){2}}
  \multiput(10.7,0)(0,4){4}%
     {\line(0,1){2}}
  \VECTOR(25,20)(10.7,0)
  \VECTOR(30,10)(12.2,0)
  \Zbox(25,21)[b]%
```

```
  {\sqrt{\omega\ped{s}}=\omega\ped{c}}
  \Zbox(31,10)[lc]{\omega\ped{s}}
\end{picture}
```

Two custom commands, `\Zbox` and `\VECTOR`, were defined in order to speed up data input. The former is short for inserting a zero dimension box at the proper coordinate, and the latter is similar to the standard `\Line` command but applied to vectors. The other unusual macro `\LINE` is completely equivalent to `\Line`; I had merely defined it before the 2009 enhancement of the `pict2e` package.

Figure 6 displays the whole diagram as scalable vector graphics, and as you can see the important messages about the filter characteristic function properties are fully and clearly expressed. This is just one example among the many such diagrams I used in my books and presentations. It was well worth the little time spent in defining the service macros. Of course I could have used the `plothandlers` module of the Ti*k*Z library, or the `tikz-3dplot` Ti*k*Z extension package, not to mention the modules associated with PSTricks. However, I saved myself the study of the 700-plus pages of Ti*k*Z documentation or a similar amount for PSTricks. The fonts in figure 6 are just the ones I designed myself for presentations; they were fully described in [1], and are available on any complete recent distribution of the TEX system.

4 Conclusion

As mentioned in the introduction, the *picture* environment is a very simple one, with but few and specific drawing commands; the documentation is so simple that less than 10 pages are sufficient. At the same time these simple commands may be used to create more complex macros and eventually result in professional drawings. Certainly this environment cannot compete with more elaborate ones, such as those provided by the packages Ti*k*Z and PSTricks; but the latter require a steep learning curve, while the former can be mastered in a few minutes. Very often the results obtained with the *picture* environment, completed by the recent enhancements provided by the `pict2e` package, are fully acceptable: it's possible to create complicated diagrams as well as simple symbols; it's possible to use this environment to place background or foreground images or symbols "wherever" on the page, even outside the margins; in [2] it is shown also how to make strange and unusual tables, Cartesian diagrams, and any sort of mix between line art and included pictures. In any case, if any lettering is placed in the drawing, it surely uses the same fonts as those used for the text, thus eliminating the usual risk that occurs when using external drawing software.

I believe that beginners would find this enhanced environment the right first step for programmed drawings; with minimum effort they can reach very good results.

References

[1] Claudio Beccari. lxfonts: LaTeX slide fonts revived. *TUGboat*, 29(2), 2008. Reprinted from ArsTEXnica #4, 2007.

[2] Claudio Beccari, editor. *Introduzione all'arte della composizione tipografica con LaTeX*. GuIt, 2011.

[3] Melchior Franz. The `crop` package, 2003. `http://mirror.ctan.org/macros/latex/contrib/crop`.

[4] H. Gäßlein, R. Niepraschk, and J. Tkadlec. The `pict2e` package, 2011. `http://mirror.ctan.org/macros/latex/contrib/pict2e`.

[5] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1990.

[6] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[7] Ian Maclaine-cross. The `curves` package, 2009. `http://mirror.ctan.org/macros/latex/contrib/curves`.

[8] Kent McPherson. The `layout` package (part of the `tools` bundle), 2000. `http://mirror.ctan.org/macros/latex/required/tools`.

[9] R. Niepraschk. The `eso-pic` package, 2010. `http://mirror.ctan.org/macros/latex/contrib/eso-pic`.

[10] M. A. Redaelli. CircuiTi*k*Z, 2011. `http://mirror.ctan.org/graphics/pgf/contrib/circuitikz`.

[11] Till Tantau. The Ti*k*Z and PGF packages, 2010. `http://mirror.ctan.org/graphics/pgf/base/doc`.

[12] Timothy van Zandt. PSTricks: PostScript macros for generic TEX, 2011. `http://mirror.ctan.org/graphics/pstricks/base/doc`.

[13] Herbert Voß. The `pst-circ` PSTricks package, 2011. `http://mirror.ctan.org/graphics/pstricks/contrib/pst-circ`.

[14] Michael Wichura. *PiCTEX*. Department of Statistics, University of Chicago, 1987.

[15] Peter Wilson and Will Robertson. The `layouts` package, 2009. `http://mirror.ctan.org/macros/latex/contrib/layouts`.

◇ Claudio Beccari
Villarbasse (TO), Italy
`claudio dot beccari (at) gmail dot com`

The apa6 LaTeX class: Challenges encountered updating to new requirements

Brian D. Beitzel

Abstract

The apa6 LaTeX class implements the document-formatting requirements of the American Psychological Association's *Publication Manual* (6th Edition). The apa6 class is an update of the outdated (and no longer maintained) apa class. This article highlights the changes and new features introduced in apa6 and describes the major obstacles I encountered during the process. Additionally, test results are presented from comparing the leading bibliographic packages for APA style.

1 Background

Social scientists are well acquainted with "APA style" (the document-formatting specifications detailed in the American Psychological Association's *Publication Manual*). Nearly all journals in the social sciences require manuscripts to be submitted in compliance with APA's *Manual*. These specifications are revised periodically, sometimes radically altering the formatting requirements — as was the case with the most recent edition.

The apa6 class is an update of the older apa class, which is no longer being maintained. The apa class formats LaTeX documents in compliance with the 5th Edition of the *Manual*. With the changes introduced in the 6th Edition of the *Manual*, the apa class was no longer adequate to meet APA specifications. Most of the changes are very welcome; however, they require significant re-adjustment from years of old habits. Users of Microsoft Word must repeatedly consult their already well-worn copies of the *Manual* while they format each successive manuscript they write. Happily for the LaTeX user, the changes simply require switching from using the apa class to using the apa6 class; but the development process entailed more than adding a "6" to the class name! This article discusses the major changes from the apa class as I updated the code to conform to 6th-Edition standards.

Like apa, apa6 has three modes that generate a different visual result when the document is compiled: `jou` mode (the default), which has a two-column, printed-journal appearance; `man` mode, which follows APA's requirements for formatting manuscripts for publication; and `doc` mode, which has a standard LaTeX-document appearance. Although some of the 6th-Edition changes (e.g., format of section headings) apply equally to `jou` and `doc` modes, in this article I will be discussing the much-more-detailed specifications from the *Manual* pertaining to `man` mode.

I will come clean here and now: I am a relatively new LaTeX user, having taught myself LaTeX and released apa6 within less than eight months. So you see, I still have a lot to learn.

2 Section headings

Probably the most sweeping change in the 6th Edition is how section headings are formatted. In the 5th and prior editions of the *Manual*, the formatting of any given heading level depended upon how many heading levels existed in the entire document. For example, in a document using five levels of headings (think of Roman-numeral outline levels) the top-level heading was formatted differently than the top-level heading in a document with only two heading levels. But the 6th Edition finally changed all that; the top-level heading is now centered, upper- and lower-cased, and boldfaced — regardless of how many heading levels the document contains. The other four heading levels have similarly specific requirements.

This change allows for simpler programming code to typeset section headings; but it also meant that I had to figure out how to format section headings using LaTeX. Fortunately, Nathaniel Smith had already done it quite well in the apa6e class, which is a limited implementation of APA style for double-spaced manuscripts. With his permission, I adopted his sectioning code and made only minimal changes.

2.1 Non-boldfaced headings

Although APA simplified the format of headings, the new requirements are still not entirely consistent: the Abstract, References, and Appendix top-level headings must *not* be boldfaced! So I had to figure out how to provide for these exceptions.

The Abstract title was rather easily taken care of by inserting `\normalfont` in the `\maketitle` definition. The Appendix titles were similarly dealt with in the `\appendix` definition.

The References heading was more of a challenge. Because apa6 allows for integration with three different bibliographic packages (biblatex, apacite, and natbib), the References heading needed to be compatible with all three. The biblatex package uses the command `\defbibheading` to specify the text for this heading; but to avoid compiling errors for documents not using biblatex, I had to check whether biblatex was loaded before issuing the `\defbibheading` command. I ended up using the etoolbox package's `\AtEndPreamble` hook to check whether the biblatex package had been loaded. Also, in `man` mode (but not `jou` or `doc` modes), the bibliography must begin

a new page, so there's another \if statement waiting to happen. Adding the \normalfont specification was a piece of cake after all that was figured out.

Similarly, apacite uses \st@rtbibsection to identify the References heading, and natbib uses \bibsection. Fortunately, defining these two commands does not throw an error if their respective packages are not loaded. The code to implement the non-boldface aspect of this heading for apacite and natbib was another gift from apa6e.

3 Author note

With the 6th Edition, APA kindly moved the Author Note (which contains contact information and acknowledgments) to the title page of the manuscript. This information used to be near the end of the manuscript where nobody noticed it.

Implementing this change mostly meant modifying the \maketitle definition, but my biggest hurdle (aside from ensuring proper paragraph indentation, handled with \indent) was figuring out the vertical placement of the Author Note heading (non-boldfaced!) and its subsequent text. I finally happened on the \vfill command which suited the situation perfectly.

4 Figure captions

Another welcome change in the 6th Edition is the placement of figure captions. In previous editions, figure captions (but not table captions) had to be presented on a separate page from their figures. The 6th Edition specifies that the captions are to be placed underneath their corresponding figures.

So, I found the part of the apa code that generated the Figure Captions page and suppressed it. Then, I ended up using the caption package to format figure captions.

5 Float placement

An additional wrinkle brought about by the 6th Edition was a change in the sequence of pages in the manuscript. Specifically, appendices are now required to be placed *after* tables and figures. This gives rise to the perplexing situation of what to do with tables and figures that are eventually to be typeset within an appendix. Does one place these floats along with the other floats from the main part of the manuscript? Or in an additional float section that follows the appendices? Or within the appendices themselves? The *Manual* is completely silent on this issue.

An APA representative acknowledged the gap in an email to me, and said that it doesn't matter which of these three solutions is followed since the *Manual* isn't clear on this point. So apa6 places appendix floats within the body of the appendix. This isn't a simple matter, however, because it means that the floats from the main part of the manuscript are delayed until a float section toward the end of the manuscript, and appendix floats are displayed at the point where they are mentioned. In other words, not all floats can be processed in the same way.

To implement this differentiation, at the beginning of the \appendix definition I inserted the \processdelayedfloats command (from the endfloat package) to flush all delayed floats prior to the appendices. Then I modified the \figure and \table definitions to delay non-appendix floats and immediately output appendix floats.

6 Floats integrated with text

When reading through a manuscript, it is a hassle to run across a reference to a table or figure and not have that table or figure handy for inspection. Prior to the 6th Edition, one only needed to look at the very end of the manuscript. But the search became more complicated with the 6th Edition because appendices now follow the tables and figures. Consequently, if there are appendices attached to a manuscript, the reader must leaf backward through the appendices to find the desired table or figure. The difficulties associated with navigating back and forth in an electronic copy of a manuscript to find tables or figures are so obvious as to need no explanation.

The floatsintext option in apa6 places all floats near the point where they are mentioned in the manuscript. Of course, such rearrangement of the manuscript violates APA guidelines and should not be used for submission to journals; but this feature can make one's own (or a colleague's) reading of drafts much more efficient.

Basically, the floatsintext option directs apa6 to handle all floats using the float package. My implementation of this feature was significantly aided by a post which Guillaume Jourjon contributed to the *LaTeX Community* Forum.

7 Reference masking

I did this one to myself. I was using the apa class to write a conference proposal and I thought how nice it would be to automagically suppress all the references to myself when submitting a paper for masked peer review.

So I chipped away at some code, did a lot of online searching and experimenting, and finally cobbled together something that seems to work. My solution was to write a series of new citation commands, which are redefinitions of the basic citation commands from

the three bibliographic packages that apa6 supports. The output of these commands replaces citations with text such as "*2 citations removed for masked review*" (in the case of two masked sources). This allows the user to specify which citations will be masked and which will not — without having to go back and edit the citation commands when the time comes to bare all.

I have yet to find a good explanation of some of the concepts operating in these functions that I wrote, but I have to say there is a sense of accomplishment in obtaining the outcome I was seeking.

8 User-selected font size

Another self-imposed difficulty was figuring out how to modify the font size of an apa6 document. I wanted to be able to provide the capability of specifying any of the standard font sizes (10pt, 11pt, 12pt) that LaTeX provides.

After several unsuccessful attempts, I gave up. Then I tried again and gave up. But it kept nagging at me. Ultimately I solved it by defining a command `\apaSix@ptsize{}` to hold the selected font size, and a comparison command, `\apaSix@noptsize{}` that was always empty. Then, with the appropriate `\DeclareOption` commands to capture the `10pt`, `11pt`, and `12pt` options, I used `\ifx` as follows for the `man` mode:

```
\ifx\apaSix@ptsize\apaSix@noptsize
  \LoadClass[12pt]{article}
\else
  \LoadClass[\apaSix@ptsize]{article}
\fi
```

This provides a default font size of 12pt, which can be overridden by the user if desired. Code similar to the above establishes 10pt as the default for `jou` mode and 11pt for `doc` mode.

9 Watermark

As I began to use my development version of apa6, new features kept creeping up on me and begged to be implemented. One of these was prominently displaying that a document was a draft version. Consequently, apa6 can now automatically load the draftwatermark package to display a "DRAFT" watermark on either the first page or on all pages of a document. This watermark can be customized with different text or font size (using the options documented with the draftwatermark package).

An unexpected situation arose, however; via the feedback of a faithful user of apa6, I discovered that when the lmodern package is not loaded, the watermark is very small. The draftwatermark documentation makes no indication of this, so I'm mentioning it here for others who may have seen similar odd renderings of this watermark package.

10 Conversion to Word

The major disadvantage of using LaTeX in the social sciences is that almost no one else does. Our journals purportedly don't accept LaTeX files (much less provide formatting classes), and they don't typically list PDF files as acceptable formats for manuscripts being submitted for review. This sparse usage of LaTeX provides challenges for (a) collaboration and (b) submission of manuscripts for publication.

This situation brings us to the need to have a way of converting a LaTeX document to a more commonly used word-processing format. I have encountered recommendations for both `htlatex` and `tth`. I have tried both and (probably because of my naivete) have been utterly unsuccessful at obtaining anything close to the PDF output produced by apa6.

Having seen a sentence online claiming that Chikrii Softlab's TeX2Word software (`http://www.chikrii.com/products/tex2word/`) was customizable in how it translated the TeX code into a Microsoft Word document, I gave it a try. Essentially, TeX2Word uses a proprietary TeX engine (thus not requiring a TeX installation) with a limited command set that closely resembles TeX. After several lengthy interchanges with their technical support, the result is an `apa6.ptex` file, included with apa6, that comes very close to replicating the PDF output of apa6. The title page, abstract page, section headings, figure and table captions, and bold and italic text are all formatted with very little need for modification (running heads are an exception). Floats must be moved from their default points within the document to their appropriate places after the reference list. The most notable exception at present is a near-total lack of bibliographic support.

11 Using apa6 with bibliography packages

Three major bibliography packages are compatible with apa6: apacite, natbib, and biblatex. I tested each of these packages against the requirements of the 6th Edition of the *Manual*. This section compares the output of these packages, highlighting inaccuracies of which authors should be aware. For details on how to use each of these packages with the apa6 class, please refer to the apa6 documentation.

11.1 Citation tests

The test cases for bibliography formatting come from the file `bibliography.bib`, which is included in the "samples" subfolder of the apa6 installation. There are several situations to examine in order to see how

Table 1: Citation test results

Test	Expected	apacite[a]	apacite-natbib[b]	biblatex-bibtex[c]	biblatex-biber[d]
1	Herbst-Damm and Kulik (2005)	pass	pass	pass	pass
2	(Herbst-Damm & Kulik, 2005)	pass	pass	pass	pass
3	(Haybron, 2008; Mayer, 2008a)	*fail*	pass	pass	pass
4a	Lassen, Steele, and Sailor (2006)	pass	pass	pass	pass
4b	Lassen et al. (2006)	pass	pass	pass	pass
5	Gilbert et al. (2004)	pass	pass	pass	pass
6a	Mayer (2008a)	pass	pass	pass	pass
6b	Mayer (2008b)	pass	pass	pass	pass
7	Mayer (in press-a, in press-b)	pass	pass	pass	pass
8	(Mayer, 2008a, 2008b)	pass	pass	pass	pass
9a	J. R. Levin and O'Donnell (2000)	pass	pass	*fail*	pass
9b	M. E. Levin and Levin (1990)	pass	pass	*fail*	*fail*
10a	(Borst, Kosslyn, et al., 2011)	pass	pass	*fail*	pass
10b	(Borst, Kievit, et al., 2011)	pass	pass	*fail*	pass
10c	(Borst, Thompson, & Kosslyn, 2011)	pass	pass	*fail*	pass
11	Franklin and Adams (2010)	pass	pass	pass	pass
12	De Waal and Grosser (2009)	*fail*	pass	pass	pass

[a] apacite version 6.01 (2012/02/25) [b] natbib version 8.31b (2010/09/13)
[c] biblatex version 1.7 (2011/11/13), biblatex-apa version 4.6 (2012/02/08), BibTeX version 0.99d
[d] biblatex version 1.7 (2011/11/13), biblatex-apa version 4.6 (2012/02/08), biber version 0.9 (2012/02/17)

well we are complying with APA requirements. The following tests are not intended to be comprehensive tests of APA citation style; rather, they cover some of the more rigorous APA-style challenges for bibliographic citation software. The number of each test corresponds to a line in Table 1 (which also summarizes the results).

1. *Joining multiple author names outside parentheses.* With a multiple-author source and when all authors are required to be listed (as opposed to the situations in Tests #4 and #5 below), the word "and" must be written out prior to the last author's name if the authors are named outside parentheses.
2. *Joining multiple author names within parentheses.* In the same situation as above, but when the authors' names are cited *inside* parentheses, the symbol "&" must be used in place of the word "and".
3. *Order citations alphabetically.* When multiple sources are cited within parentheses, they must be sorted in the same order in which they appear in the reference list at the end of the manuscript. For this test, citations were purposely entered in reverse alphabetical order.
4. *Truncating 3–5 author names.* When there are 3–5 authors, all authors' names are listed for the first citation; subsequent citations list only the first author's name followed by "et al."
5. *Truncating six or more author names.* When there are more than six authors, only the first author should ever be listed, followed by "et al.".
6. *Same author(s), same year.* When different articles have the identical author(s) in the same year, the year must be followed by "a", "b", etc.
7. *Same author(s), in press.* When different in-press articles have identical author(s), the year must be given as "in press-a", "in press-b", etc.
8. *Same author(s), different articles.* When citing two or more articles by the same author(s) within parentheses, the author name(s) need not be repeated.
9. *Different first authors, same last name.* When two first authors have the same last name, their initials must be given to clarify which one is being cited.
10. *Multiple authors, same year.* When two or more articles have a subset of the same authors in the same order, all citations must give as many author names as necessary to make the citation unique. Note that the "al." in "et al." is plural and therefore must replace at least two names.
11. *Suppress name suffixes.* The suffix of author names (e.g., "Jr.") should not be included when citing their work in the body of the text.
12. *Capitalizing initial lower-case names.* If the first word in a sentence is an author name that begins

with a lower-case letter (e.g., "de Waal"), the name should nonetheless be capitalized.

11.2 Results of citation tests

The results of these citation tests are shown in Table 1. First, it should be noted that most of these packages handle basic citations very well. Only one of them passed all 12 tests, but two others did very well. Additionally, not all failures in Table 1 are equally egregious; for example, the single biblatex-biber failure (Test #9b) will never cause confusion as to which source is being cited.

Before we look at the results, I wish to applaud the developers of apacite and biblatex-apa for responding to my initial citation test results and modifying their packages to better comply with 6th Edition requirements. There are now many fewer failed tests in this series than there were when I first ran these tests with then-current versions of these packages only a few months ago.

The results for each package are summarized next. No reviewer or journal editor will ever remark on "the amazing accuracy of your citations"; but comments to the opposite effect may be encountered. Unfortunately, we therefore need to focus on the non-compliance here rather than what each package does right.

11.2.1 apacite

The apacite package was loaded using the `apacite` option when loading the apa6 class, like this:

```
\documentclass[jou,apacite]{apa6}
```

There were two apacite errors: (a) in Test #3, the references were not sorted alphabetically within the parentheses; and (b) in Test #12, the prefix "De" could not be capitalized because apacite does not provide any capitalization command.

To overcome the failures of Test #3, one must manually sequence the parenthetical citations; this is entirely feasible but does require a certain level of alertness on the part of the author. There is no cure for the failure of Test #12 without adding on the natbib package (see next section).

11.2.2 apacite-natbib

Both the apacite package and the natbib package were loaded implicitly using the `natbib` option when calling the apa6 class:

```
\documentclass[jou,natbib]{apa6}
```

There were no apacite-natbib errors, thanks to some clever programming by the apacite developer, Erik Meijer. The natbib package does not contain a bibliographic style; therefore, apacite is required when using natbib with apa6. The apacite package contains directives that load natbib and appropriately modify natbib commands to conform to 6th Edition requirements. The apa6 user simply needs to specify the `natbib` option to load and configure both of these packages properly.

11.2.3 biblatex with bibtex

The biblatex package was loaded with the following options specified:

```
\usepackage[style=apa,sortcites=true,
        sorting=nyt]{biblatex}
```

There were five biblatex (with bibtex) errors: (a) in Test #9a, the first author's initials were not given; this is a serious error, explicitly violating APA requirements because another author has the same surname; (b) in Test #9b, the same problem was encountered; (c–e) in Tests #10a, #10b, and #10c, the references were identified as "(Borst et al., 2011a)," "(Borst et al., 2011b)," and "(Borst et al., 2011c)." Although the Test #10 results do not cause confusion in identifying the intended source, this format does not conform to APA requirements.

11.2.4 biblatex with biber

The biblatex package was loaded with the following options specified:

```
\usepackage[style=apa,sortcites=true,
    sorting=nyt,backend=biber]{biblatex}
```

There was only one minor biblatex (with biber) error: In Test #9b, the second author's initials were given (only first authors' initials are required when there are multiple authors with the same surname).

11.3 Conclusions from citation tests

For APA-style citations, the apacite-natbib and the biblatex-biber solutions are clearly the most competent; the only error was relatively minor and would never cause confusion as to which source is being cited (biblatex-bibtex).

Time for a personal admission: For several months after learning about biber I was daunted by using it because for some reason I thought that once I converted to biber I was more or less committing myself to it for life. However, that is not so; to use biber, there are no changes required in the `.bib` file (although some advantages can be gained from a few label changes). All it takes is including the `backend=biber` option when loading the biblatex package. It could hardly be simpler!

11.4 Reference tests

The in-text citations are only part of the battle; formatting the reference list correctly is the other critical test for a bibliography package. I checked

the reference list output from each package against 6th Edition requirements and found no errors that could not have been predicted by the results of the citation tests already described.

11.4.1 apacite

The apacite package produced a perfect reference list for my sample sources.

11.4.2 apacite-natbib

The apacite-natbib solution also had no errors in the reference list.

11.4.3 biblatex with bibtex

Strangely, biblatex (with bibtex) erred in sorting two of the references: Borst, Kosslyn, et al., 2011 was listed prior to Borst, Kievit, et al., 2011. I don't have a clue as to why this would be; I even tried switching the BibTeX keys but the sorting remained unchanged.

Additionally there is the problem of the three Borst references having "a", "b", and "c" (respectively) appended to their publication dates. This is not necessary because the author lists for these three references are unique.

11.4.4 biblatex with biber

The biblatex-biber solution also produced a perfect reference list.

11.4.5 Attention to the details

To conclude, let's show off a 6th Edition formatting requirement that all four of these bibliography solutions can now do. D. Gilbert and eight other individuals published an article in 2004. Check out the reference below and you will see that the first six authors are listed, followed by an ellipsis, followed by the final author. This way of handling more than six authors in the reference list is a new stipulation in the 6th Edition.

> Gilbert, D., McClernon, J., Rabinovich, N., Sugai, C., Plath, L., Asgaard, G., ... Botros, N. (2004). Effects of quitting smoking on EEG activation and attention last for more than 31 days and are more severe with stress, dependence, DRD2 A1 allele, and depressive traits. *Nicotine & Tobacco Research, 6*(2), 249–267. doi:10.1080/14622200410001676305

11.5 EndNote results

For comparison with a leading commercial bibliographic manager, I also subjected the latest version of EndNote (X5.0.1) to each of these tests. EndNote failed citation tests #10c (substituting "et al." in place of only one author name, the same error as biblatex) and #12 (with no capitalization available for a lower-case name, the same flaw as apacite). There were no errors on the reference list.

12 Conclusion

Although I didn't know exactly what I was getting myself into when I began tinkering with the apa code, I'm glad I tackled this project. I've learned a lot, and with the help of others in the TeX community, apa6 can be even better in the future.

◇ Brian D. Beitzel
129 Fitzelle
SUNY Oneonta
Oneonta, NY 13820 USA
brian (at) edpsych dot net
http://www.edpsych.net/brian/

Glisterings

Peter Wilson

> Sound like bels, and shine like lanternes.
> Thunder in words and glister in works.
>
> *School of Abuse*, Stephen Gosson

The aim of this column is to provide odd hints or small pieces of code that might help in solving a problem or two while hopefully not making things worse through any errors of mine.

> Something old, something new,
> Something borrowed, something blue.
>
> *Traditional: Advice to the Bride*

This issue's column reiterates two items from the electronic TEXMAG journal, years ago.

> I didn't go to the moon, I went much further — for time is the longest distance between two places.
>
> *The Glass Menagerie*,
> Tennessee Williams

1 Timelines

This is a slightly edited version of an article that Don Hosek wrote for TEXMAG, titled *Timelines with plain TEX and LATEX* [1].

In one issue of *TUGboat* (Vol. 8, No. 2), there was a query for a macro to draw timelines in TEX. At the time, I had just finished writing DVIview and was waiting for bugs to surface and my paycheck to arrive with little else to do, so I decided to tackle the problem.

To make the problem more interesting, I decided to make the macro work in both LATEX *and* plain TEX. A sample input file should look something like:

```
%%% LaTeX sample
\documentclass{article}
\usepackage{timeline}
\def\TeXMaG{\TeX
   M\kern-.1667em\lower.5ex\hbox{A}%
    \kern-.2267emG}
\begin{document}
This is a timeline of the history of
the first year of \TeXMaG.
\begin{timeline}{2in}(0,180)
\optrule
\item[12]{Jan. 24}{No. 1}
\item[33]{Mar. 6}{No. 2}
\item[43]{Mar. 25}{No. 3}
\item[81]{May. 13}{No. 4}
\item[102]{Jun. 25}{No. 5}
\item[132]{Aug. 24}{No. 6}
\item[160]{Oct. 10}{No. 7}
\item[179]{Dec. 31}{To be}
\end{timeline}%% Must be a comment here!!!
If text immediately follows the end of the
timeline then a comment is required
otherwise there is an extraneous space at the
start of the text. A blank line following the
end acts normally, whether or not there has
been a comment.
\end{document}
```

And something similar in plain TEX. In the example above, I used sort keys to control the spacing between entries. I also could have had a timeline whose entries looked like this:

```
\begin{timeline}{1.5in}(1750,1900)
\optrule
\item{1773}{The Tea Party}
\item{1812}{War of 1812}
\item{1849}{Gold rush of '49}
\item[1862]{1862--5}{Civil War}
\item{1876}{Little Big Horn} % added by PW
\end{timeline}
```

where the dates themselves control the placement. Note that the entry for the Civil War uses a sort key to allow the year to be 1862–5. This was a pretty big problem. The comments in the code below give a fair idea of how to *use* the macro; the remainder of this section will deal with how the macros themselves *work*.

First of all, it helps to have some idea of how `\begin{...}` and `\end{...}` work in LATEX. To view it in a simplified form, when the command `\begin{FOO}` is invoked, LATEX issues the commands `\begingroup` followed by `\FOO`; similarly, `\end{FOO}` issues the commands `\endFOO` followed by `\endgroup`. Therefore, to allow an environment to function in plain TEX, all we need to do is include an extra grouping with `\FOO...\endFOO` and provide copies of any LATEX internal macros used by the environment. Both of these tasks are fairly simple, and in the timeline macros, the only LATEX internal macro called is `\@ifnextchar` (this is a very handy macro for many reasons, and gives some insight into the mysteries of LATEX).

`\@ifnextchar` is called with the general form:

```
\@ifnextchar X{YES}{NO}
```

The timeline macros use this for `\item` to check to see if the optional argument (enclosed in `[]`s) is present. If the next character after `\@ifnextchar` matches `X` (`X` cannot be a space) then `YES` is executed, otherwise `NO` is executed. In the specific case here, this is done with the call:

```
\@ifnextchar[\@item\@itemnosrtkey
```

This calls `\@item` if the optional argument is present, and `\@itemnosrtkey` if it isn't. In addition, the character that is tested remains in the input stream, so `\@item` has a parameter list that looks like
`\@item[#1]#2#3`
rather than
`\@item#1]#2#3`.

The definition of `\item` for the timeline macros is kept local so it won't interfere with other uses of that control sequence name by either plain or LaTeX. The main work of this is done by `\@item`, which takes three arguments: the first argument is used to determine the vertical placement of the timeline item, the second argument is the nominal date and the third a description. `\@itemnosrtkey` calls `\@item` using the nominal date as the first argument as well.

The placement of the item on the timeline is determined by taking the date number (first parameter) and converting to a number between 0 and the length of the timeline as specified with the arguments to the `\timeline` (`\begin{timeline}`) macro. This number is then multiplied by 1/65536 times the length of the timeline as specified by the user. The factor of 1/65536 prevents an arithmetic overflow from occuring at the cost of reducing accuracy (measurements are only kept accurate to one point). Finally, this number is divided by the length of the range of date number values and then multiplied by 65536 which gives a dimension specifying how far down from the top of the timeline the entry should be placed.

The actual placement is accomplished with the `\dlap` macro from the toolbox of TeXMaG Vol. 1 No. 3 [by Barbara Beeton]. By placing the necessary text after a vertical `\kern`, inside a vertical lap, we are able to print information anywhere on the timeline without changing our vertical position. This does have the disadvantage of using a lot of box memory and may run into problems with very complicated timelines, but it seemed like a good idea at the time.

The final interesting facet of the macros is the (simple) way that two entries that are close together are resolved. After an entry is printed, the vertical dimension specifying its placement is stored in the dimen register `\itwashere`. When the next entry is to be printed, the current vertical placement is compared to `\itwashere`; if the difference is less than 12pt, and the entry would normally be placed on the left, then the entry is printed on the right. Otherwise it is printed on the left. This algorithim works well for two closely placed entries but fails for three closely placed entries (the two on the left will likely overlap).

This is a timeline of the history of the first year of TeXMaG.

Jan. 24 No. 1 •
Mar. 6 No. 2 •
• **Mar. 25** No. 3
May. 13 No. 4 •
Jun. 25 No. 5 •
Aug. 24 No. 6 •
Oct. 10 No. 7 •
Dec. 31 To be •

If text immediately follows the end of the timeline then a comment is required otherwise there is an extraneous space at the start of the text. A blank line following the end acts normally, whether or not there has been a comment.

Figure 1: First timeline

The macros presented work for simple timelines, but probably will be deficient for more complex timelines. Hopefully, this explanation of the macros will help in customizing them for your own purpose, or in writing a timeline macro of your own.

```
%%% File: timeline.sty
%%% Works with either LaTeX or plain TeX
%%%
%%% In LaTeX:
%%% \begin{timeline}{length}(start,stop)
%%%   . . .
%%% \end{timeline}
%%%
%%% in plain TeX
%%% \timeline{length}(start,stop)
%%%   . . .
%%% \endtimeline
%%% in between the two, we may have:
%%% \item{date}{description}
%%% \item[sortkey]{date}{description}
%%% \optrule
%%%
%%% the options to timeline are:
%%%     length --- The amount of vertical space
%%%        that the timeline should use.
%%%     (start,stop) --- indicate the range of
%%%        the timeline. All dates or sortkeys
%%%        should lie in the range [start,stop]
%%%
%%% \item without the sort key expects date to
%%%       be a number (such as a year).
%%% \item with the sort key expects the sort
%%%       key to be a number; date can be
%%%       anything. This can be used for log
%%%       scale timelines or dates that
%%%       include months or days.
```

```
%%% putting \optrule inside of the timeline
%%%       environment will cause a vertical
%%%       rule to be drawn down the center
%%%       of the timeline.

\catcode`\@=11     % Pretend @ is a letter
\newcount\startat    \newcount\tllength
\newdimen\putithere  \newdimen\itwasthere
\newcount\scr@tchi   \newdimen\scr@tchii

% A vertically centered lap
\long\def\ylap#1{\vbox to \z@{\vss#1\vss}}

% Vertical 'laps'; cf. \llap and \rlap
\long\def\ulap#1{\vbox to \z@{\vss#1}}
\long\def\dlap#1{\vbox to \z@{#1\vss}}

\def\timeline#1(#2,#3){%
  \ifvmode\else\par\fi$$\vbox to#1\bgroup
          % The \vbox command is
          % surrounded by $$..$$ to make it
          % fit in well with paragraphs.
  \offinterlineskip
  \startat=#2\tllength=#3
    \advance\tllength by-\startat
% \tllength should be the total length of
%  the timeline.
  \def\item{\@ifnextchar[\@item\@itemnosrtkey}
  \def\@item[##1]##2##3{\scr@tchi=##1
    \advance\scr@tchi by-\startat
    \putithere=#1
    \divide\putithere by 65536 % avoid overflow
% only remain accurate to 1pt in the
% next set of calculations
    \multiply\putithere by \scr@tchi
    \divide\putithere by\tllength
    \multiply\putithere by 65536
% Now \putithere has how far
% down we should go for this item.
    \scr@tchii=\putithere
    \advance\scr@tchii by -\itwasthere
    \ifdim\scr@tchii<12pt
      \ifx\lrswitch L
        \@putright{\putithere}{##2}{##3}
      \else
        \@putleft{\putithere}{##2}{##3}
      \fi
    \else
      \@putleft{\putithere}{##2}{##3}
    \fi
    \itwasthere=\putithere}
  \def\@itemnosrtkey##1##2{%
    \@item[##1]{##1}{##2}}
  \def\@putright##1##2##3{\dlap
    {\kern##1\centerline
      {\rlap
        {\ $\bullet$\hskip1.5em{\bf ##2}
         \ ##3}}}
    \let\lrswitch=R}
  \def\@putleft##1##2##3{\dlap
    {\kern##1\centerline
      {\llap
        {{\bf ##2} \ ##3\hskip1.5em$\bullet$
          \ }}}
    \let\lrswitch=L}
  \def\optrule{\dlap
    {\centerline
% This calculation is kept local
      {\dimen0=#1 \advance\dimen0 by 6pt
        \vrule depth \dimen0 height-6pt}}}

% Put the extra \vskip in a \vbox to hide
% it from the math gods.
\def\endtimeline{%
  \vfill\egroup\vbox{\vskip\baselineskip}$$}
\ifx\@latexerr\undefined
  \def\@ifnextchar#1#2#3{%
    \let\@tempe #1
    \def\@tempa{#2}\def\@tempb{#3}
    \futurelet\@tempc\@ifnch}
  \def\@ifnch{%
    \ifx \@tempc \@sptoken
      \let\@tempd\@xifnch
    \else
      \ifx \@tempc \@tempe
        \let\@tempd\@tempa
      \else
        \let\@tempd\@tempb
      \fi
    \fi \@tempd}
% NOTE: the following hacking must precede
%       the definition of \: as math medium
%       space.
% make \@sptoken a space token
  \def\:{\let\@sptoken= } \:
  \def\:{\@xifnch}
  \expandafter\def\: {\futurelet\@tempc\@ifnch}
\catcode`\@=12 % Stop pretending @ is a letter
\fi
\endinput
```

Using the above code, the result of the initial example is in Figure 1 and the second is in Figure 2.

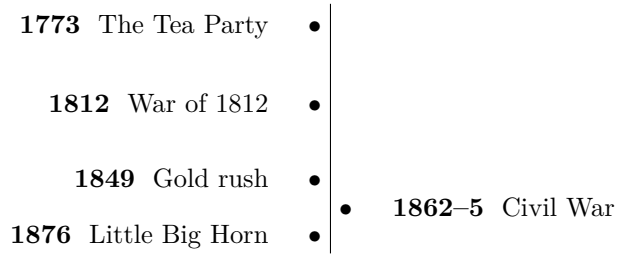


Figure 2: Second timeline

> Gaul as a whole is divided into three parts.
>
> *De Bello Gallico*, Julius Caesar

2 Parsing a filename

This is another (slightly edited) article from a later issue of TEXMAG [2].

Sometimes it is nice to be able to use the information in a `filename.tex` as information in a particular document. For example, suppose I wanted to typeset TEXMAG on real paper, and be able to have the volume and issue numbers read from the title of the file that TEX was processing, and subsequently assigned to tokens for use in the document, perhaps in a header. Say my file was named `TEXMAG-5-1.TEX`. The following would isolate the 5 and the 1 for use within the TEX document:

```
% This particular idea was developed by our
% chief consultant Dr. John McClain

\newtoks \volumenumber
\newtoks \issuenumber

\def\parse#1-#2-#3-{\global\volumenumber={#2}
                    \global\issuenumber={#3}}

\expandafter\parse\jobname-

%%% for a TeX headline
\headline={Volume \the\volumenumber,
            Number \the\issuenumber
\hfil page \folio}
% end of macro
```

Notice the `\jobname` contains the name of the file (without any extension, see *The TEXbook*, p. 213). The `\expandafter` allows you to piece apart this token into its volume and number. We also had to chose a special delimiter which would conform to filename standards and be a legal parameter delimiter in TEX. A space would not have worked as a legal file name. A hyphen was our best choice. When you test this, remember that the filename must conform to the parameter specs of `\parse` (in this case, two hyphens, i.e., `XXXX-N-N.TEX`).

The essence of the code in the TEXMAG article is the `\parse` macro. The `\jobname` of the document you are now reading is '`tb103glister`', which does not match the requirements of `\parse`. The following code demonstrates that macros based on `\parse` can work with names other than `\jobname`, provided that they expand into the expected format.

For instance:

```
\newcommand*{\jname}{glisten-n16-v3.tex}
\newtoks \pwfirstsub
\newtoks \pwsecondsub
\def\parse#1-#2-#3.#4-{%
    \global\pwfirstsub={#2}
    \global\pwsecondsub={#3}}
\newcommand*{\parsit}[1]{\expandafter\parse#1-}

\verb?\parsit{\jname}? \parsit{\jname}
  File \jname\ with: \\
Number \the\pwfirstsub,
and Version \the\pwsecondsub.
```

And the result of the above code is:

`\parsit{\jname}` File glisten-n16-v3.tex with: Number n16, and Version v3.

The basic idea of `\parse` can be applied to any multipart string that has well-defined delimiters between the parts.

References

[1] Don Hosek. Timelines with plain TEX and LATEX. *TEXMAG*, 1(7), October 1987. `http://mirror.ctan.org/digests/tex-mag/v1.n7`.

[2] John McClain. The toolbox. TEXMAG. `http://mirror.ctan.org/digests/tex-mag/v5.n1`.

◇ Peter Wilson
12 Sovereign Close
Kenilworth, CV8 1SQ
UK
herries dot press (at)
earthlink dot net

Some LaTeX 2ε tips and tricks (V)

Luca Merciadri

1 Introduction

This time, as usual, we shall see some LaTeX hints (numbered according to the following sections):

2. Numbering paragraphs,
3. Incorporating MATLAB graphics,
4. Incorporating MATLAB code,
5. Customizing an index.

2 Numbering paragraphs

2.1 Example

Here is an example of numbering paragraphs:

The Title
1 One line of text that is long enough to wrap as a paragraph that is long enough to wrap
2 Second batch of text that is long enough to wrap as a paragraph that is long enough to wrap as a paragraph.
3 Still more lines of text that are long enough to wrap as a paragraph that is long enough to wrap as a paragraph.

2.2 Code

This can be done with the following code:

```
\newcounter{vcount}
\def\Header#1{\medskip%
  \hbox{\bfseries #1}%
  \setcounter{vcount}{1}%
  \everypar{\arabic{vcount}%
  \stepcounter{vcount}\ }%
}
```

You can then use

```
\Header{The Title}
First paragraph.

Second paragraph. [...]
```

3 Incorporating MATLAB graphics

MATLAB can output graphics, and of course one may want to incorporate them into a LaTeX document. This can be achieved easily, and it produces a better-looking document, because of a more coherent presentation. There are two essentially different approaches: `laprint` and Ti*k*Z/PGF-related ones — `Matfig2PGF` and `matlab2tikz`. Thanks to Marc van Dongen for telling me about the second of these. We will describe each separately.

3.1 LaPrint

For this, you need `laprint.m`. According to [9], once `LaPrint` has been launched into MATLAB, it can perform the following tasks:

- Replace all occurrences of text in the MATLAB figure by tags,
- Save the modified figure in PostScript format (`eps` file),
- Create a `tex` file with commands from the LaTeX `psfrag` package to replace the tags by the original text and to call the PostScript file.

Let's assume you have typed

```
>> set(0,'defaulttextinterpreter','none')
>> figure(1),clf
>> plot([1 2])
>> ylabel('A straight line')
```

where "`>>`" denotes MATLAB's prompt. Let's then type (assuming `laprint.m` is in your current MATLAB working directory)

```
>> laprint
```

`LaPrint` thus asks you the "Number of Figure to be Saved" and "Basename of Files to be Created". You can modify several options, then click on "`Go!`." The `laprint` script will then create two files: an `eps` one and a `tex` one.

The `tex` file can be included into LaTeX documents using the packages `graphicx`, `color` and `psfrag`. Thus, if you let "Basename of Files to be Created" to "`unnamed`", a simple `tex` file showing your graphics will be generated, and will have content like this:

```
\documentclass{article}
\usepackage{graphicx,color,psfrag}
\begin{document}
\input{unnamed}
\end{document}
```

For other pieces of information (such as how to give a predetermined size to your graphics, ...), do not hesitate to read [9].

3.2 Matfig2PGF

`matfig2pgf` converts a MATLAB figure to the Portable Graphics Format (PGF). This PGF file can be included in a LaTeX document. Once `matfig2pgf` has been launched in MATLAB, you just need to generate your plot in MATLAB, and then invoke `matfig2pgf` using

```
>> matfig2pgf('myfile.pgf')
```

where `>>` is MATLAB's prompt and `myfile.pgf` is the output file. You can now write your `.tex` document according to the following minimal structure:

```
\documentclass{article}
\usepackage{pgf}
\usepackage{pgffor}
\usepgflibrary{plothandlers}

% Or, for older PGF versions (<= 1.01)
%\usepackage{pgf}
%\usepackage{pgffor}
%\usepackage{pgflibraryplothandlers}

\begin{document}
  \begin{figure}
    \centering
    \input{myfile.pgf}
    \caption{Figure created by Matfig2PGF}
  \end{figure}
\end{document}
```

This is an easy way to put a MATLAB figure into a LaTeX document. For more information, try typing

```
>> help matfig2pgf
```

in MATLAB.

3.3 Matlab2Tikz

A third way to achieve this is to use `matlab2tikz`. Once `matlab2tikz` has been launched in MATLAB, again you just need to generate your plot in MATLAB, and then invoke `matlab2tikz` using

```
>> matlab2tikz('myfile.tikz');
```

where `>>` is MATLAB's prompt and `myfile.tikz` is the output file. You can now write your `.tex` document according to the following minimal structure:

```
\documentclass{article}
\usepackage{tikz}
\usepackage{pgfplots}
\begin{document}
\input{myfile.tikz}
\end{document}
```

For more information, please have a look at [1].

You may note that you can do all these things by using SageTeX, but it is a little bit less straightforward. It is also possible that you might have to use more than one approach (especially coupling `laprint` with the two other approaches). For example, the plot resultig from a `spectrogram` command in MATLAB can only be included in a LaTeX document with `laprint`.

4 Incorporating MATLAB code

To typeset MATLAB code ([8, 10]), one good approach is to use the `listings` package together with `mcode`.[1] Thus, you may put

```
\usepackage{listings}
\usepackage[bw,numbered,framed,final]{mcode}
```

in the preamble of your document. The following options are available for `mcode`:

- `bw` is useful if you intend to print the document,
- `numbered` is useful if you want line numbers to be written before each line of code,
- `framed` is useful if you want a frame around the source code blocks,
- `final` is useful if you have "globally" set the `draft` option, as `listings` will not, in such a case, output the code at all. That forces it to do so anyway.

You can then include a MATLAB source file using

```
\lstinputlisting{/path/to/yourmfile.m}
```

or placing snippets of source code in a `lstlisting` environment. For example, you would then do

```
\begin{lstlisting}
% Example of Matlab code for calculating
% hypotenuse
% § $c=\sqrt{a^{2}+b^{2}}$ §
a = 3;
b = 4;
c = sqrt(a^2+b^2);
\end{lstlisting}
```

Note that "§" allow you to "escape" from LaTeX mode. As a result, you are not obliged to pass lots of parameters to `listings` using `lstset`.

This will give a better presentation than using `lstlisting` together with a declaration like

```
\lstset{language=MATLAB,basicstyle=\small%
\ttfamily,showstringspaces=false,%
numbers=left,commentstyle=\itshape,%
backgroundcolor=\color{white},%
stepnumber=2,numbersep=5pt,%
escapeinside={(*@}{@*)}}
```

5 Customizing an index

5.1 Standard customizations

When generating an index with `makeindex`, one can create a `perso.ist` file with "customizations". For example:

```
heading_prefix "{\\bfseries\\hfil "
heading_suffix "\\hfil}\\nopagebreak\n"
headings_flag 1
delim_0 "\\dotfill"
delim_1 "\\dotfill"
delim_2 "\\dotfill"
```

[1] Note that this package may be downloaded at `http://files.myopera.com/locksley90/blog/mcode.sty`, at `http://web.mit.edu/~paul_s/www/14.170/matlab/mcode.sty` or even at [8].

This writes the first alphabet symbol in bold font, and uses dots as delimiters. This file is generally used jointly with `makeindex` using

```
makeindex -s perso.ist filename.idx
```

where `filename.idx` has been created by executing `latex` on `filename.tex`.

5.2 French tricks

If your document is in French, you could ask for "Symboles" at the place of "Symbols" and "Nombres" at the place of "Numbers." This is achieved by appending

```
symhead_positive   "Symboles"
symhead_negative   "symboles"
numhead_positive   "Nombres"
numhead_negative   "nombres"
```

to the previous code.

5.3 Insensitive letter sort

If you want, for example, an insensitive letter sort for letter A, you may use, according to [3]:

```
sort_rule "A" "a"
```

You can then repeat this rule for every letter.

5.4 Special letter sort

For TEX-style umlaut-macros, you may use, according to [2]:

```
sort_rule "\\\"A" "ae"
sort_rule "\\\"a" "ae"
sort_rule "\\\"O" "oe"
sort_rule "\\\"o" "oe"
sort_rule "\\\"U" "ue"
sort_rule "\\\"u" "ue"
sort_rule "\\ss({})?" "ss"
```

5.5 Math formulae sort

If you use fancy constructs such as

```
\index{log@\texttt{log}}
```

you may use, according to [5]:

```
% first remove enclosing '$'-characters
*merge_rule "\$(.*)\$" "\1"

% function-name macros will be sorted like
% the function they stand for
merge_rule "\\log" "log"
merge_rule "\\lim" "lim"
% etc.
```

5.6 Greek letter sort

For Greek letters, you may use, according to [5]:

```
% the pronounciation of Greek letters
% decides their sort order
merge_rule "\\alpha" "alpha"
merge_rule "\\beta" "beta"
merge_rule "\\gamma" "gamma"
% etc.
```

5.7 Special characters sort

According to [6], you may use

```
% special characters come first
sort_rule "\." "\b\."
sort_rule "\:" "\b\:"
sort_rule "\," "\b\,"
% etc.
```

to handle special characters correctly.

5.8 Last refinements

If the commands `\LaTeX` and `\TeX` are not correctly handled by your `makeindex`, you may use, according to [4, 7]:

```
merge_rule "\\LaTeX" "LaTeX"
merge_rule "\\TeX" "TeX"
```

References

[1] Universiteit Antwerpen. `matlab2tikz`, 2009. `http://win.ua.ac.be/~nschloe/content/matlab2tikz`.

[2] Gabor Herr. `din.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/din.ist`.

[3] Gabor Herr. `icase.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/icase.ist`.

[4] Gabor Herr. `latex.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/latex.ist`.

[5] Gabor Herr. `math.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/math.ist`.

[6] Gabor Herr. `puncts.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/puncts.ist`.

[7] Gabor Herr. `tex.ist`, 1991. `http://mirror.ctan.org/indexing/makeindex/ist/tex.ist`.

[8] Florian Knorn. M-code LaTeX Package, 2009. `http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package`.

[9] Arno Linnemann. LaPrint Users Guide (LaPrint Version 3.16), 2004. `http://www.uni-kassel.de/fb16/rat/matlab/laprint/laprintdoc.ps`.

[10] Locksley. How to include MATLAB source code in a LaTeX document, 2009. `http://my.opera.com/locksley90/blog/2008/02/25/how-to-include-matlab-source-code-in-a-latex-document`.

◇ Luca Merciadri
University of Liège
`Luca.Merciadri (at) student dot ulg dot ac dot be`
`http://www.student.montefiore.ulg.ac.be/~merciadri/`

LaTeX News

Issue 6, June 2011

A key aim of releasing 'stable' LaTeX3 material to CTAN is to allow users to benefit from new ideas *now*, and also to raise the profile of usable LaTeX3 ideas. This is clearly being successful, with xparse being of particular utility to end users. This increase in interest has been particularly notable on the new TeX.SX Q&A site.

The LaTeX3 Team expands

Raising interest in LaTeX3 developments has inevitably led to feedback on cases where the code base has required attention. It has also attracted new programmers to using LaTeX3 ideas, some more than others! Bruno Le Floch has over the past few months made many useful contributions to LaTeX3, and we are very pleased that he has recently joined the LaTeX3 Project.

Bruno has taken a particular interest in improving the performance and reliability of the expl3 language. This has already resulted in new implementations for the `prop` and `seq` data types. At the same time, he has identified and fixed several edge-case issues in core expl3 macros.

The 'Big Bang'

In parallel to Bruno's improvements, Joseph Wright initiated a series of 'Big Bang' improvements to LaTeX3. The aim of the Big Bang was to address a number of long-standing issues with the LaTeX3 code base. Development has taken place over many years, with the status of some of the resulting code being less than clear, even to members of The LaTeX3 Project! At the same time, different conventions had been applied to different parts of the code, which made reading some of the code rather 'interesting'. A key part of the Big Bang has been to address these issues, cleaning up the existing code and ensuring that the status of each part is clear.

The arrangement of LaTeX3 code is now the same in the development repository and on CTAN, and splits the code into three parts.

l3kernel The core of LaTeX3, code which is expected to be used in a LaTeX3 kernel in more or less the current form. Currently, this part is made up of the LaTeX3 programming layer, expl3.

l3packages LaTeX 2$_\varepsilon$ packages making use of LaTeX3 concepts and with stable interfaces. The xparse and xtemplate packages are the core of this area. While many of the *ideas* explored here may eventually appear in a LaTeX3 kernel, the interfaces here are tied to LaTeX 2$_\varepsilon$.

l3experimental LaTeX 2$_\varepsilon$ packages which explore more experimental LaTeX3 ideas, and which may see interface changes as development continues. Over time, we expect code to move from this area to either l3kernel or l3packages, as appropriate.

In addition to these release areas, the development code also features a l3trial section for exploring code ideas. Code in l3trial may be used to improve or replace other parts of LaTeX3, or may simply be dropped!

As well as these improvements to the *code* used in LaTeX3, much of the documentation for expl3 has been made more precise as part of the Big Bang. This means that `source3.pdf` is now rather longer than it was previously, but also should mean that many of the inaccuracies in earlier versions have been removed. Of course, we are very pleased to receive suggestions for further improvement.

LaTeX3 on GitHub

The core development repository for LaTeX3 is held in an SVN repository, which is publicly viewable *via* the Project website. However, this interface misses out on some of the 'bells and whistles' of newer code-hosting sites such as GitHub and BitBucket. We have therefore established a mirror of the master repository on GitHub.[1] This is kept in synchronisation with the main SVN repository by Will Robertson (or at least by his laptop!).

The GitHub mirror offers several useful features for people who wish to follow the LaTeX3 code changes. GitHub offers facilities such as highlighted differences and notification of changes. It also makes it possible for non-Team members to submit patches for LaTeX3 as 'pull requests' on GitHub.

As well as offering a convenient interface to the LaTeX3 code, the GitHub site also includes an issue database.[2] Given the very active nature of LaTeX3 development, and the transitory nature of many of the issues, this provides a better approach to tracking issues than the main LaTeX bug database.[3] Developers and users are therefore asked to report any issues with LaTeX3 code *via* the GitHub database, rather than on the main Project homepage. Discussion on the `LaTeX-L` mailing list is also encouraged.

Next steps

The 'Big Bang' involves making a number of changes to expl3 function names, and is likely to break at least some third-party code. As a result, the updates will not appear on the TeX Live 2011 DVD release, but will instead be added to TeX Live once regular updates restart (probably August).

Bruno is working on a significant overhaul of the l3fp floating-point unit for LaTeX3. He has developed an approach which allows expandable parsing of

[1] `http://github.com/latex3/svn-mirror`
[2] `http://github.com/latex3/svn-mirror/issues`
[3] `http://www.latex-project.org/bugs.html`

floating-point expressions, which will eventually allow syntax such as

```
\fp_parse:n { 3 * 4 ( ln(5) + 1 ) }
```

This will result in some changes in the interface for floating-point numbers, but we feel that the long-term benefit is worth a small amount of recoding in other areas.

Joseph has completed documentation of the xgalley module, and this is currently being discussed. Joseph is hoping to move on to implement other more visible ideas based on the xtemplate concept over the next few months.

LaTeX News

Issue 7, February 2012

After the 'Big Bang'

The last LaTeX3 News gave details of the 'Big Bang', in which the team have revised the layout and coverage of the LaTeX3 codebase. This process has made the status of different modules clearer, so that both the team themselves and everyone else know what is going on.

The 'Big Bang' changes were not shipped to CTAN until after the TeX Live 2011 freeze, as we did not want to end up with a DVD containing badly broken code. The update went to CTAN soon after TeX Live 2011 shipped, and has now propagated around the world. The new package naming (l3kernel, l3packages and l3experimental) has caused some surprises for a small number of users, but there have not been any major issues with the changes at the code level.

The 'Big Bang' has attracted attention from programmers outside of the LaTeX3 team, with useful feedback arriving on the `LaTeX-L` list and TeX.SX, in particular. One area that this has highlighted is the need to document carefully when changes to the 'stable' parts of the LaTeX3 codebase occur. All changes to l3kernel now come with an explicit date for the change in the documentation, which means that programmers can check exactly when the features they want were introduced.

Another key part of supporting LaTeX3 use beyond the team is making it easy to check on the version of LaTeX3 installed. To support that, the file date of the main expl3 package is now set each time there is a release of the LaTeX3 material to CTAN. This means that the LaTeX 2$_\varepsilon$ `\@ifpackagelater` test can be used reliably to detect if the installed version of LaTeX3 is going to supply the functions that a programmer is using.

Deforming boxes

Additions to both the LaTeX3 stable material and more experimental modules continue. Joseph Wright has been working on adding 'native' drivers for LaTeX3 to support box transformations. These allow box rotation, clipping and scaling with the drivers `dvips`, `xdvipdfmx` and direct PDF output.

The development of clipping support for the `xdvipdfmx` driver has also allowed us to suggest improvements to the LaTeX 2$_\varepsilon$ graphics drivers, enabling clipping with the XeTeX engine.

A TeX-based regex engine

Bruno Le Floch has been improving the efficiency and robustness of a number of LaTeX3 functions. Most notably, he has created a purely TeX-based regular expression (regex) system for LaTeX3. This is currently experimental, but is already proving useful and will hopefully stabilise over the coming months.

Bruno's regex system works with all of the supported engines (pdfTeX, XeTeX and LuaTeX). He has implemented the core ideas of standard regex systems, along with some TeX-specifics to allow matching and replacing the content of token lists by category code.

xparse improves

The xparse module has been overhauled, making the internal code more efficient and adding additional argument types. This has also allowed us to deal with a number of internal bugs, meaning that argument grabbing is now more reliable.

The argument grabbers themselves have been reworked so that in the event of an error, the user will normally get a meaningful message from TeX rather than one pointing to xparse internal function names. This should help in tracking down erroneous input in real documents.

The galley

As detailed in the last issue, work on the galley module has been continuing. Discussion of Joseph's reimplementation of the galley concepts highlighted some important areas to work on, with the nature of the template concept being particularly significant.

More work is still needed to finalise the galley concepts, but it is clear that some of this will require feedback from other areas. Joseph therefore hopes to finish work on the current round of galley improvements by the end of February, and to return to them once some other areas have been addressed.

Relationships between document items

The TUG 2011 meeting took place in October in India. Frank Mittelbach spoke there about ideas for describing the design relationship between document elements. These ideas allow a document designer to specify the design of a document element based on its context within a document, and progress in this area will likely lead to an extension in the xtemplate system.

User-friendly web utilities for generating LaTeX output and METAPOST graphics

Troy Henderson

Abstract

There are several facets of the creation of LaTeX documents and METAPOST graphics that deter users from initially trying both LaTeX and METAPOST. These include the basic structure of the source files, the compilation of the source files, and the conversion of the output to a desired format. Furthermore, many TeX users often desire to create 2D and 3D graphs of functions for inclusion into their documents. Many of these types of graphs require considerable amounts of source code to create professional quality graphics, and this is yet another deterrent for those who might otherwise consider using METAPOST. This article introduces several free web utilities that aim to eliminate each of these obstacles and describes the usage and methods of these utilities.

1 LaTeX Previewer

LaTeX is a powerful typesetting system, but there are several reasons that discourage document preparers from using (in fact, even trying) LaTeX. Probably the most common reason is the fact that LaTeX is not a WYSIWYG word processor such as most document preparers are accustomed to using. As a result, these preparers might be interested in trying LaTeX while at the same time they might also be overwhelmed by LaTeX due to some of its characteristics. Several of these characteristics could include, for example, LaTeX's large installation size, relatively complex (source) document structure for beginners, compilation process, and lack of real-time previewing of LaTeX output.

Several years ago, the LaTeX Previewer was created to address these issues and provide beginners with a user-friendly interface to LaTeX that did not require users to download and install LaTeX and did not require any knowledge of the LaTeX document structure or compilation process. The LaTeX Previewer can be (freely) used by visiting

`http://www.tlhiv.org/ltxpreview`

Figure 1 shows the initial display for the Previewer. Beginners simply type source code and preview the corresponding output by selecting the Preview button. If a user requires the inclusion of particular LaTeX packages, then the Packages button can be used to accesses a user-friendly interface for adding and removing different packages. If the desired package is not listed, then the user is advised to send an email (and is provided a link) to request additional packages. Figure 2 illustrates an example of how the source code is rendered. If the user makes a mistake in typing the LaTeX source code and if a compilation error occurs, the user is informed of this error and is encouraged to view the compilation output log using the Log button. If compilation is successful, the (cropped) output can be downloaded in a variety of different formats including LaTeX source, Encapsulated PostScript (EPS), Portable Document Format (PDF), Portable Network Graphics (PNG), Scalable Vector Graphic (SVG), and Adobe Shock-Wave Flash (SWF).

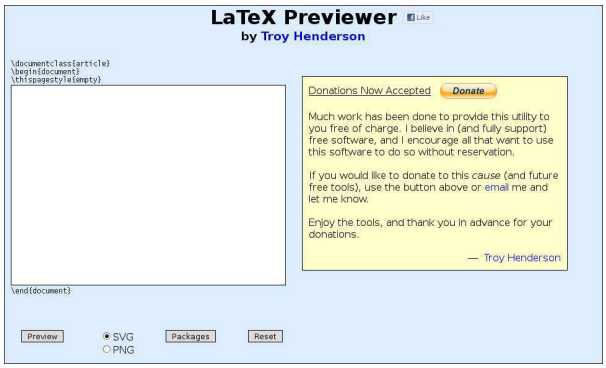


Figure 1: LaTeX Previewer Initial Display

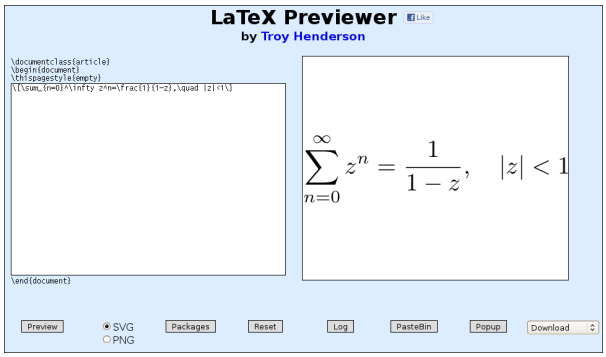


Figure 2: LaTeX Previewer Example Run

There is also a PasteBin button for sharing the source code with others as well as a Popup button for opening a larger window to view the output. Finally, the user's web browser is detected and it is automatically determined if proper SVG support is available, and if so, the default rendered output format is SVG. If the user's browser does not have proper SVG support, then the failsafe PNG rendered output is used. Furthermore, the user can override the automatic output rendering format by manually selecting either the SVG or PNG radio button.

2 METAPOST Previewer

METAPOST, like LaTeX, can be unwelcoming to beginners who wish to create professional quality graphics. These beginners, when first witnessing METAPOST examples, often wish to start creating their own graphics by typing source code. A typical METAPOST source file often needs `prologues` set appropriately to ensure that fonts used throughout the graphic are embedded in the resulting output. Furthermore, since many METAPOST beginners are experienced in LaTeX, they typically would like to typeset labels in METAPOST using LaTeX. In order to accomplish this, several commands are required in the METAPOST source. Since a METAPOST source file can accommodate multiple figures, the output naming scheme can be quite confusing to beginners, and it may also be unclear that each output is in fact an EPS file (even though each output filename extension may not be `.eps`).

These issues, as well as several others, were the primary motivation for creating the METAPOST Previewer. The METAPOST Previewer was introduced in *TUGboat* [2] in 2007, but it has since been given more features and made more user-friendly. Like the LaTeX Previewer, the goal was to provide a user-friendly interface to METAPOST that makes these issues transparent to beginners. Also, like the LaTeX Previewer, the METAPOST Previewer can be (freely) used by visiting

`http://www.tlhiv.org/mppreview`

The user interface for the METAPOST Previewer is virtually identical to that of the LaTeX Previewer, and an example is shown in Figure 3. The only noticeable difference between the two Previewers is that the Packages button for the METAPOST Previewer provides the user with the ability to select both LaTeX *and* METAPOST packages. For example, the user can choose the LaTeX package `arev` to have labels typeset in the Arev Sans (a derivative of Bitstream Vera Sans) font and also choose the METAPOST package `boxes` (for drawing a variety of boxes in METAPOST).

3 Previewer framework

Both Previewers have a user interface that is written in HTML (the main markup language for web pages) and CSS (a style sheet language for adjusting the appearance of web pages). This user interface provides dynamic interaction using JavaScript (a client-side scripting language used to enhance websites). When users enter source code into the Previewer and select Preview, the source code is posted to a CGI script (a standard method for processing HTML form data) written in Perl. The CGI script creates a LaTeX/METAPOST source file that contains the preamble presented on the Previewer's page, any user-selected packages to be included in the preamble, as well as the user-provided source code. The CGI script then compiles the source file with `latex` or `mpost` using a *halt on error* interaction method in order to determine whether compilation errors occur. If no errors occur, the LaTeX/METAPOST output (DVI/EPS) is converted to the on-screen display format, returned to the Previewer, and the output is then available for conversion to the downloadable formats listed above. Figure 4 illustrates the steps used to convert the source to each available format.

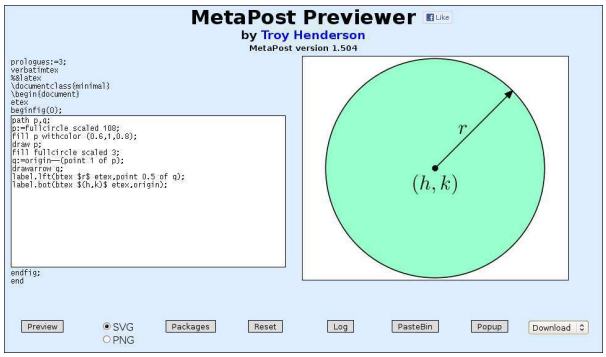


Figure 3: METAPOST Previewer Example

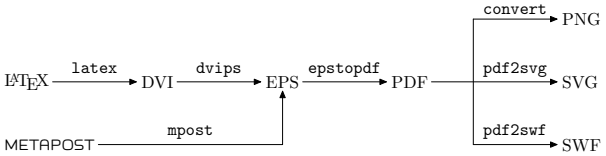


Figure 4: Previewer Conversion Process

The first few commands, namely `latex`, `dvips`, `mpost`, and `epstopdf` are included in most TeX distributions, and these sequential commands provide PDF output which is used as the basis for all other downloadable formats. The `convert` command is provided by ImageMagick [3] which (quoting from its website) "is a software suite to create, edit, compose, or convert bitmap images". Furthermore, `pdf2svg` is a utility by David Barton [1] which uses Poppler and Cairo to convert PDF documents/graphics to SVG format. Finally, `pdf2swf` is part of the SWFTools [5] suite and is a PDF to SWF converter.

4 Function Grapher

A standard type of graphic that TeX users often include in their documents are graphs of functions, curves, and surfaces. There are a variety of commercial applications available that can generate these

graphs, and each of these applications requires at least a moderate level of expertise in not only generating the graph but also exporting the graph to a TEX-friendly format. Furthermore, these graphs often appear *out of place* in TEX documents since the fonts may not be consistent with the document font and the line widths of the curves may not be consistent with default TEX document rules (lines). Of course, these issues can be addressed, but typically even more expertise is needed to accomplish this consistency. Finally, these commercial applications can cost anywhere between $100 to $2,500 (USD) which may be impractical for many users. There is an abundance of free (many open-source) applications which can produce these graphs as well, but, typically, an even greater level of expertise than for the commercial applications is needed to produce high-quality graphs.

The primary reason for creating the Function Grapher was to provide a free utility that produces publication-quality graphs with a user-friendly interface. The user interface of the Function Grapher is similar to that of the Previewers, and its output is designed to mimic MATLAB's [4] graphs with several personal preferences incorporated. Like the Previewers, the Function Grapher can be (freely) used by visiting

`http://www.tlhiv.org/mpgraph`

Figure 5 illustrates a preview of the Function Grapher, and shows the polar plot of $r(t) = \sin(8t/5)$ with $0 \leq t \leq 10\pi$.

Currently, the Function Grapher can graph (up to) three functions of a single variable simultaneously, parametric plane curves of a single variable, polar curves, contour plots of a function of two variables, slope fields of first order ordinary differential equations, parametric space curves, surface plots of (up to) three functions of two variables simultaneously, and parametric surfaces. Each type of graph has two predefined examples (accessed using the corresponding Example button) that can be used to illustrate syntax and output quality. The Options button provides users with the ability to adjust the appearance of each graph. Users can adjust the graph's aspect ratio, color scheme, axis labels, axis and grid visibility for each type of graph and can adjust mesh visibility for 3D surfaces. The output can be downloaded in a variety of formats for stand-alone graphics or for insertion into TEX documents.

5 Function Grapher framework

Like the Previewers, the user interface for the Function Grapher is written in HTML and CSS, and dynamic interaction is accomplished using JavaScript.

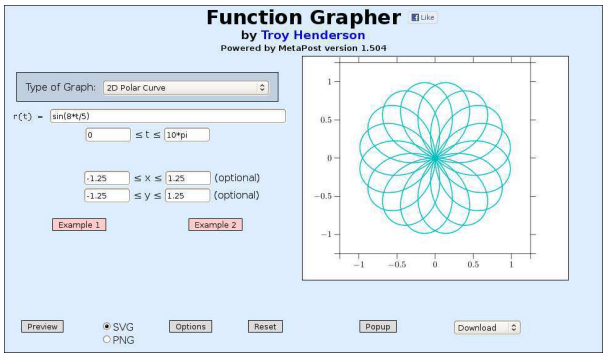


Figure 5: Function Grapher Example

Also, like the Previewers, when users enter functions to graph and select Preview, the function information is posted to a (Perl) CGI script which processes the HTML form data. The CGI script creates a uniform partition of each independent variable's interval and then evaluates the function(s) at the nodes of this partition. The density of the partition is chosen to balance quality with computation time, and the extreme values of each function are determined from these evaluations.

The tick marks for both the independent and the dependent variables are then computed using Wilkinson's algorithm [6], and clipping of the curve or surface is performed if the user specifies a range on the dependent variable(s) which is more narrow than the computed extreme values. The 3D graphics are represented using an orthographic projection of the xyz-space in which the uv-projection plane is orthogonal to a viewpoint vector w on the unit sphere. That is, $w = \langle \cos\theta\cos\phi, \sin\theta\cos\phi, \sin\phi \rangle$ where $-\pi \leq \theta < \pi$ and $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$. The uv-projection plane is chosen so that the z-axis is projected to the v-axis (i.e., the z-direction is always drawn vertically). Both θ and ϕ can be specified using the Options button, and have default values of $\theta = -127.5°$ and $\phi = 20°$. These values can be changed manually by editing the corresponding HTML form fields or semi-automatically by clicking the rotation arrows overlaid on the previewed graphic. The final steps in creating each graph are accomplished by the CGI script writing a METAPOST source file and then generating output using the process described in Figure 4.

6 Function Grapher examples

This section gives several example graphs generated by the Function Grapher. Figure 6 is the graph of

three functions of a single variable, namely

$$\begin{aligned} f(x) &= \sin(2\pi x) \\ g(x) &= \sin\big(2\pi x - \tfrac{2\pi}{3}\big) \\ h(x) &= \sin\big(2\pi x + \tfrac{2\pi}{3}\big) \end{aligned}$$

representing three phase power.

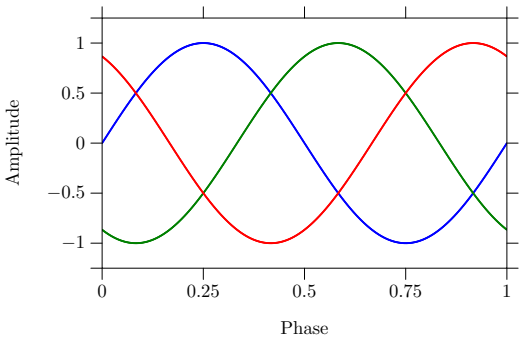


Figure 6: Three Phase Power

Figure 7 shows several contours (level curves) of the function

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\tfrac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \tfrac{1}{3}e^{-(x+1)^2-y^2}.$$

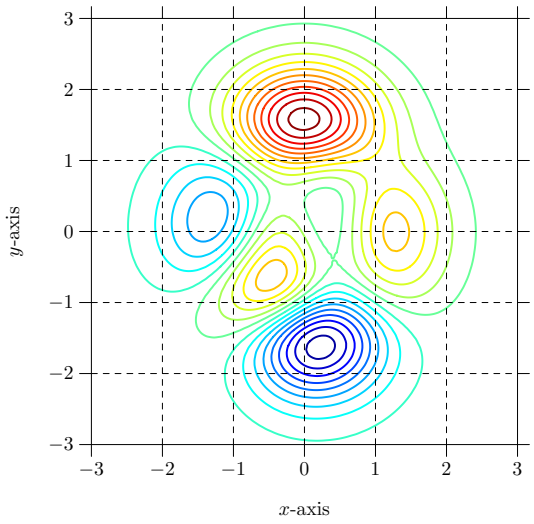


Figure 7: Contour (Level Curve) Plot

Figure 8 is the slope field plot of the ordinary differential equation

$$\begin{cases} \dfrac{dy}{dx} = x + y \\ y(0) = 0 \end{cases}$$

where the numerical solution is approximated using a fourth order Runge-Kutta method, and Figure 9 is the surface plot of the function

$$f(x,y) = e^{-x^2-y^2}.$$

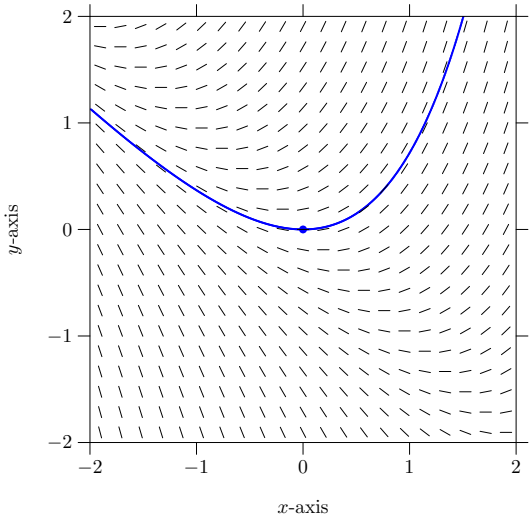


Figure 8: ODE Slope Field Plot

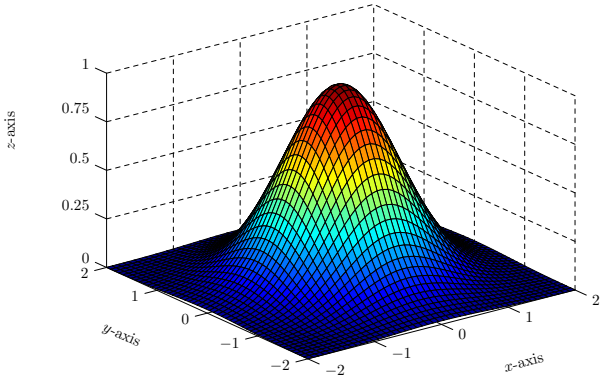


Figure 9: Surface Plot of Function of Two Variables

Figure 10 is the graph of the parametric (torus) surface parameterized by

$$\begin{aligned} x(s,t) &= (3+\cos s)\cos t \\ y(s,t) &= (3+\cos s)\sin t \\ z(s,t) &= \sin s, \end{aligned}$$

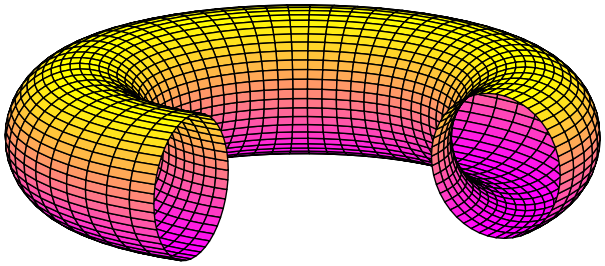


Figure 10: Surface Plot of Torus

Figure 11 is the graph of the parametric (heart) surface parameterized by

$$\begin{aligned} x(s,t) &= \tfrac{1338}{557} r(t)\cos(t)\cos(s) \\ y(s,t) &= 2\sin(s) \\ z(s,t) &= \tfrac{1338}{557}\left[r(t)\sin(t) + \tfrac{970}{557}\right]\cos(s) \end{aligned}$$

where

$$r(t) = 2 - 2\sin(t) + \frac{\sin(t)\sqrt{|\cos(t)|}}{\sin(t) + \frac{7}{5}}$$

with $-\frac{\pi}{2} \le s \le \frac{\pi}{2}$ and $-\pi \le t < \pi$.

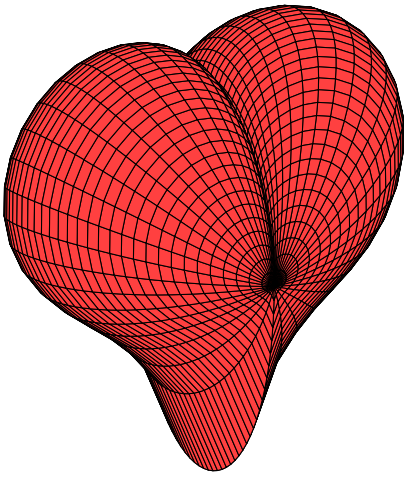


Figure 11: Surface Plot of a 3D Heart

Finally, Figure 12 is the graph of the parametric space curve parameterized by

$$\begin{aligned} x(t) &= \cos(6t) \\ y(t) &= \sin(6t) \\ z(t) &= t. \end{aligned}$$

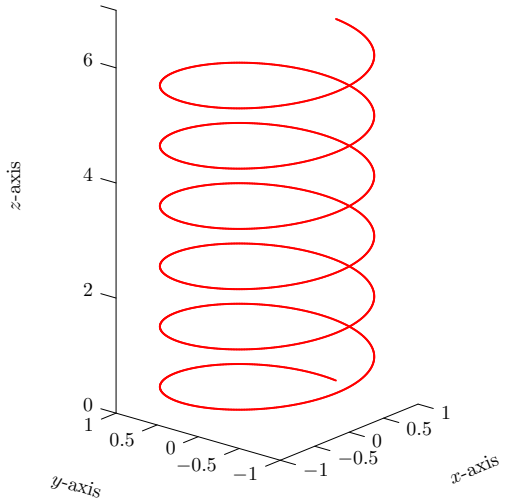


Figure 12: Plot of a Helix Space Curve

7 Conclusion

It is the author's desire that these web-based utilities will simplify the introduction to LaTeX and METAPOST and, as a consequence, help users prepare professional-quality manuscripts (both text layout and graphics). These utilities will continue to be available for free, and users are encouraged to direct feature requests and any other type of feedback to the addresses below.

References

[1] David Barton. `http://www.cityinthesky.co.uk/opensource/pdf2svg`.

[2] Troy Henderson. A beginner's guide to MetaPost for creating high-quality graphics. *TUGboat*, 28(1):85–90, 2007.

[3] ImageMagick. `http://www.imagemagick.org`.

[4] Mathworks. `http://www.mathworks.com/products/matlab`.

[5] SWFTools. `http://www.swftools.org`.

[6] Leland Wilkinson. *The Grammar of Graphics*. Springer-Verlag New York, Inc., 2005.

◇ Troy Henderson
Department of Mathematics
University of Mobile
5735 College Parkway
Mobile, AL 36613 USA
`thenderson (at) umobile dot edu`
`http://www.tlhiv.org`

TEX on Windows: MiKTEX or TEX Live?

Joseph Wright

On Windows, there are two actively-developed free TEX systems with similar coverage: MiKTEX (Schenk, 2011) and TEX Live (TEX Users Group, 2011). The good news is that there is a lot of similarity between the two systems, so for most users both systems are equally usable, and (LA)TEX documents are portable between them. However, there are differences and depending on what you need these might be important.

- The default settings install everything for TEX Live, but only a minimal set of packages for MiKTEX. MiKTEX will then install extra packages 'on the fly', while TEX Live does not (there is a package to do that in TEX Live, but it is aimed at GNU/Linux users). Install-on-the-fly is useful if space is limited, but is more problematic on server setups. So this is very much a feature whose usefulness depends on your circumstances. Of course, there is nothing to stop you from installing everything with MiKTEX.
- The xindy program (Schrod, 2010) is available only in TEX Live. For those of you not familiar with it, xindy is an index processor, and is much more capable of dealing with multi-lingual situations than MakeIndex. If you need xindy, TEX Live is the way to go.
- MiKTEX is very much a Windows tool set, while TEX Live comes from a Unix background. This shows up from time to time in the way TEX Live is administered, and the fact that the TEX Live GUI is written based on Perl rather than as a 'native' Windows application.
- As TEX Live is the basis of MacTEX, and is the TEX system for Unix, if you work cross-platform and want an identical system on all of your machines, then TEX Live is the way to go.

A reminder that MiKTEX and TEX Live are not the only choices. W32TEX (Kakuto, 2012) is popular in the far east. As well as being a TEX system in its own right, it is the source of the Windows binaries for TEX Live, and TEX Live acquires more CJK support from it every year. For users focussed on ConTEXt, ConTEXt standalone (Pragma ADE, 2012) is probably the best way to go (it uses the W32TEX binaries on Windows). There are also the commercial options, for example BaKoMa TEX (BaKoMa Soft., 2011) or PCTEX (Personal TEX, Inc., 2011). However, for most users it comes down to a choice between the 'big two'.

References

BaKoMa Soft. "BaKoMa TEX 9.77".
http://www.bakoma-tex.com/, 2011.

Kakuto, A. "W32TEX".
http://w32tex.org/, 2012.

Personal TEX, Inc. "PCTEX 6".
http://www.pctex.com/, 2011.

Pragma ADE. "ConTEXt standalone".
http://wiki.contextgarden.net/ConTeXt_Standalone, 2012.

Schenk, C. "MiKTEX 2.9".
http://www.miktex.org/, 2011.

Schrod, J. "xindy 2.4".
http://xindy.sourceforge.net/, 2010.

TEX Users Group. "TEX Live 2011".
http://www.tug.org/texlive/, 2011.

◇ Joseph Wright
Morning Star
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
joseph.wright (at)
morningstar2.co.uk

Generating barcodes with LuaTEX

Patrick Gundlach

Abstract

TEX is great at typesetting, Lua is great with calculations. When we combine those two, we can do complex typesetting tasks easily. In this article, I present a way to create GTIN-13 barcodes (also known as EAN 13) with LuaTEX. The aim of this article is to present how to call Lua from TEX, some basic Lua programming and two ways in which Lua and TEX can interact.

1 Introduction

There are several ways to generate barcodes from TEX: one is the PSTricks barcode package; a few packages rely on special fonts; and one I found generates barcodes with vertical rules, but the source is not suitable for beginners and therefore rather hard to extend. The Lua solution I present is supposed to be easier to understand for non-TEX programmers, but this is subjective, of course. For the purpose of this demonstration, only EAN 13 barcodes are handled, optionally calculating the checksum (the last digit) if the requested barcode has only twelve digits. This is an example output of our program:

The LaTeX interface should be as simple as this:

```
\documentclass{article}
\usepackage{ltxbarcode}
\begin{document}
\barcode{424200251816}
% or -- with checksum:
\barcode{4242002518169}
\end{document}
```

The "glue" style file `ltxbarcode.sty` is short as well. You can place it in the same directory as the main LaTeX file above:

```
\ProvidesPackage{ltxbarcode}

\directlua{require("ltxbarcode")}

\newcommand\barcode[1]{%
  \directlua{
    ltxbarcode.generate_barcode("#1")
  }}
```

The package loads the file `ltxbarcode.lua`, which is given and explained in full detail below. `require` appends the extension `.lua` by itself. Then it defines the command `\barcode`, whose sole task is to jump into Lua mode (`\directlua`) and call the Lua function `generate_barcode()`, passing along the argument given to the LaTeX command. The prefixed namespace `ltxbarcode` is automatically created with `require()` at the beginning.

There is a small pitfall here. You would normally write this code as:

```
\newcommand\barcode[1]{%
  \directlua{
   ltxbarcode.generate_barcode(
     "\luatexluaescapestring{#1}"
     )}}
```

to protect the Lua string from the first macro argument (`#1`) containing characters that might possibly break the Lua parser. If the macro argument contains, for example, a double quote character, Lua will see it as the end of the quoted string and choke on the rest of the argument. Since we are only passing ordinary digits, the string is safe in our code above. (Protection is a good idea, though.) The long name from above (`\luatexluaescapestring`) is the command `\luaescapestring` found in the reference manual prefixed with `luatex` to avoid name clashes. The only command from LuaTEX not prefixed is `\directlua`. This is only valid for TEXLive's current LuaLaTeX format. The plain LuaTEX format in TEX Live has all commands unprefixed. I fervently hope that other distributions behave exactly the same.

Before we have a close look at our Lua module (the file we load in our LaTeX package), let's take an excursion on how to communicate between the Lua mode and LuaTEX.

2 From Lua to TEX

Passing arguments from TEX to Lua is easy, as seen above; e.g., the call to `generate_barcode()`. But the other way is a bit more interesting, as one has to keep in mind when the code gets executed. The Lua code in `\directlua` gets executed the moment LuaTEX finds the closing brace of that command. It will then be replaced by the special buffers that this command fills. Example:

```
  \directlua{
    tex.sprint("\\hbox{%")
    tex.sprint("hello world%")
    tex.sprint("}")
   }
```

and the TEX code

```
\hbox{%
hello world%
}
```

are more or less equivalent. Thus, strings can be split into smaller chunks and automatically concatenated (with line endings as separators) automatically at the end of the `\directlua` call. What is not possible, though, is the following:

```
\directlua{
  tex.sprint("\\setbox0\\hbox{hello world!}")
  % does NOT work because box 0 is not set yet:
  tex.sprint(
    string.format(
      "The width of box 0 is now \%d",
        tex.box[0].width ))
}
```

Inside `\directlua` it is not possible to mix TEX and Lua calculations like this, because the TEX value is not known until the end of the `\directlua` call. So you cannot operate on the box dimensions before TEX typesets that box. Keep this in mind as we compare the two approaches I will show now.

3 Solution one: `tex.sprint()`

The first approach is to calculate the barcode itself (this is a simple routine) and construct a set of `\hbox`, `\vrule` and `\kern` commands with the Lua function `tex.sprint()`. Before we dive into the main function, we start with the head of the Lua file (named `ltxbarcode.lua`). Most Lua modules start with a call to `module()`. We make all but the main functions *local*, meaning that they are only visible inside the module.

```
module(...,package.seeall)
local add_checksum_if_necessary, mkpattern,
     split_number, calculate_unit, pattern_to_wd_dp
```

Now comes the heart of the module. We use the method described above to generate a sequence of TEX commands that get executed right after the `\directlua` call. The idea is to draw the barcode with vrules and kerns and add the digits below in a separate box.

```
function generate_barcode( str )
  -- If we only pass 12 digits, the 13th will be added.
  str = add_checksum_if_necessary(str)

  -- The smallest bar/gap is 1/7th the width of a digit.
  -- It is font dependent.
  local u = calculate_unit()

  -- We start with the hbox for the bars:
  tex.sprint(
    [[\newbox\barcodebox\setbox\barcodebox\hbox{%]]
     )

  -- The pattern is a string of digits that represent
  -- the width of a bar or a gap. 0 is a special marker
  -- for a longer bar of width 1. The widths are
  -- multiplied by 1/7th of the width of a digit, because
  -- the sum of the widths for a single digit add up to 7.
  -- A sample pattern starts with:
  -- 801032111123121321132311321110101320...
  -- See function mkpattern() for a detailed explanation.
  local pattern = mkpattern(str)

  -- For each element in the pattern we generate a gap or
  -- a bar of the width denoted by that element. A depth
  -- >0 is used for the bars in the middle and both sides.
  -- This is technically not necessary, but added to have
  -- visually pleasing barcodes.
  local wd,dp -- width and depth of a bar
  for i=1,string.len(pattern) do
    wd,dp = pattern_to_wd_dp(pattern,i)
    -- The even elements are the vertical bars (vrules),
    -- the odd ones are the gaps (kerns).
    if i % 2 == 0 then
      tex.sprint(string.format(
        [[\vrule width %dsp height 2cm depth %s]],
        wd * u,dp))
    else
      tex.sprint(string.format(
          [[\kern %dsp]],wd * u))
    end
  end
  -- We now have the hbox with the bars and
  -- add the hbox with the digits.
  tex.sprint([[}\vbox{\hsize\wd\barcodebox \box\
       barcodebox\kern -1.7mm\hbox{%]])

  -- The digits below the barcode are split into three
  -- groups: one in front of the first bar, the first
  -- half of the other digits are left of the center
  -- bar, and the remaining digits are to the right
  -- of the center bar.
  local first,second,third = split_number(str)
  tex.sprint(string.format(
     [[%s\kern %dsp %s\kern %dsp%s}]],
     first, 5 * u, second, 4 * u, third ))
end
```

The main function uses several helper functions. One of them calculates the width of the smallest bar and the smallest gap, which is exactly 1/7th the width of a digit. We make use of LuaTEX's font library where we can get access to the current font. The glyph number 48 is the digit zero; theoretically, this is encoding-dependent, but in practice it works in all cases.

```
function calculate_unit()
  -- The relative widths of a digit represented by the
  -- barcode add up to 7.
  local currentfont = font.fonts[font.current()]
  local digit_zero = currentfont.characters[48]
  return digit_zero.width / 7
end
```

The next function determines the width and the depth of a vertical rule. The height is fixed (we could have made that customizable, but the reader should be left with some task to do).

```
function pattern_to_wd_dp( pattern,pos )
  local wd,dp
```

```
  wd = tonumber(string.sub(pattern,pos,pos))
  if wd == 0 then
    dp = "2mm"
    wd = 1
  else
    dp = "0mm"
  end
  return wd,dp
end
```

The calculation of the checksum is straightforward. We sum up all the digits, every other digit is multiplied by 3 (counted from the last digit backwards) and the checksum is the amount you need to add to get to the next multiple of 10. The sum is only calculated if not given by the user. (A future version could check a user-supplied value.)

```
function add_checksum_if_necessary( str )
  if string.len(str) == 13 then
    return str
  end

  local sum = 0
  local len = string.len(str)
  for i=len,1,-1 do
    if (len - i ) % 2 == 0 then
      sum = sum + tonumber(string.sub(str,i,i)) * 3
    else
      sum = sum + tonumber(string.sub(str,i,i))
    end
  end
  local checksum = (10 - sum % 10) % 10
  return str .. tostring(checksum)
end
```

The following pattern generation is the heart of the algorithm. The barcode is divided into smaller parts where two bars and two gaps represent a single digit. The widths of these vary between "one" and "four", multiplied by any sensible width. The widths for a single digit add up to 7 of these units and are expressed by a simple pattern such as *2221* for the digit 1. The first digit in a barcode is not represented by a bar–gap pair, but rather encoded in the representation of the next six digits. If, for example, the first digit is a *1*, the third, fifth and sixth "digits" have to be reversed. See the array `mirror_t` in the code below. In the example above the reversed pattern is *1222*. We add some space to the left of the barcode for the first digit and also mark the left and right edge with the special mandatory pattern *111*. Actually it is *010* which we recognize later to increase the length of these bars.

```
function mkpattern( str )
  -- These are the digits represented by the bars.
  -- 3211 for example means a gap of three units,
  -- a bar two units wide, another gap of width one
  -- and a bar of width one.
  local digits_t = {"3211","2221","2122","1411",
            "1132","1231","1114","1312","1213","3112"
       }

  -- The first digit is encoded by the appearance of the
  -- next six digits. A value of 1 means that the
  -- generated gaps/bars are to be inverted.
  local mirror_t = {"------","--1-11","--11-1",
              "--111-","-1--11","-11--1","-111--",
                "-1-1-1","-1-11-","-11-1-"}

  -- Convert the digit string into an array.
  local number = {}
  for i=1,string.len(str) do
    number[i] = tonumber(string.sub(str,i,i))
  end

  -- The first digit in a barcode determines how the
  -- next six digit patterns are displayed.
  local prefix = table.remove(number,1)
  local mirror_str = mirror_t[prefix + 1]

  -- The variable pattern will hold the constructed
  -- pattern. We start with a gap that is wide enough
  -- for the first digit in the barcode and the special
  -- code 111, here written as 010 as a signal to
  -- create longer rules later.
  local pattern = "8010"
  local digits_str

  for i=1,#number do
    digits_str = digits_t[number[i] + 1]
    if string.sub(mirror_str,i,i) == "1" then
      digits_str = string.reverse(digits_str)
    end
    pattern = pattern .. digits_str
    -- The middle two bars.
    if i==6 then pattern = pattern .. "10101" end
  end
  -- Append the right 111 pattern as above.
  return pattern .. "010"
end
```

The last function splits the barcode into three parts so we can display the digits below the barcode with some gaps in between.

```
function split_number( str )
  return string.match(
    str,"(%d)(%d%d%d%d%d%d)(%d%d%d%d%d%d)"
    )
end
```

The net result of this code is a TEX string like this:

```
\newbox\barcodebox\setbox\barcodebox\hbox{%
\kern 374492sp
\vrule width 46811sp height 2cm depth 2mm
\kern 46811sp
...
\vrule width 46811sp height 2cm depth 2mm
}\vbox{\hsize\wd\barcodebox\box\barcodebox\kern -1.7
     mm\hbox{%
8\kern 234057sp 008940\kern 187246sp027004}}
```

This is what TEX sees after the closing brace of `\directlua`. While this solution works fine in our small example, it can get a bit tedious, because of the string passing and the necessity to escape all occurrences of the well-known *funny* TEX chars such as `%` and others. Luckily with LuaTEX, our new swiss army knife in the TEX world, we have another approach to that problem.

4 Solution two: direct typesetting with low-level nodes

The other approach to that problem looks like using a sledge-hammer to crack a nut. But once one becomes used to it and some helper functions defined, this solution is well-suited for many tasks when we are using Lua for program logic. The idea is to create the fundamental data structures TEX uses internally for representing the typeset material: a *node*. A node can represent a glyph, a rule, a glue, a whatsit and all other items we know from *The TEXbook*. The typeset digit '0' for example could be represented by a table with these entries:

entry	value
`id`	37
`char`	48
`font`	15
`lang`	0

There are other optional entries in that table, but only the *prev* and the *next* entries are necessary for building a more complex data structure. The table above can be constructed from Lua like this:

```
n = node.new("glyph") -- internal id: 37
n.char = 48
n.font = 15
n.lang = 0
```

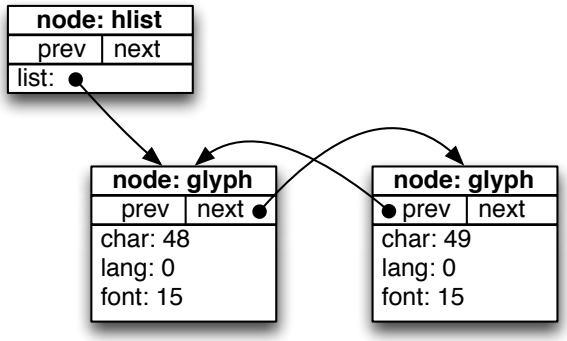
To construct a horizontal box with the digit created above a call to `node.hpack()` is sufficient:

```
hbox = node.hpack(n)
```

Which is the same as `\hbox{0}` except that the box is only kept in TEX's memory and not put into the PDF. It gets more complex when you want more than one item to be placed in a box. You then need to create the nodes and chain them together into a list. Every node has *prev* and *next* table entries which are to be set to the predecessor and successor nodes. So in the case of the two digits 0 and 1 placed in a horizontal box, it would look like:

```
digit_0 = node.new("glyph")
digit_1 = node.new("glyph")
-- not shown: fill the tables as above
digit_0.next = digit_1
digit_1.prev = digit_0
hbox = node.hpack(digit_0)
```

The result is a data structure that can be visualized by the following graphic:



The *list* entry of the hlist (hbox) points to the *node list* starting with the digit 0. The idea for our second approach is to create a node list that represents the vertical bars and gaps (rule and kern nodes) and digits. We create a few more helper functions as well as the new main function:

```
local add_to_nodelist, mkrule, mkkern, mkglyph
function generate_barcode_lua( str )
  str = add_checksum_if_necessary(str)

  local u = calculate_unit()
  local nodelist

  -- The even elements are the rules,
  -- the odd ones are the gaps.
  local pattern = mkpattern(str)
  local wd,dp
  for i=1,string.len(pattern) do
    wd,dp = pattern_to_wd_dp(pattern,i)
    if i % 2 == 0 then
      nodelist = add_to_nodelist(
        nodelist,mkrule(
          wd * u,tex.sp("2cm"),tex.sp(dp)))
    else
      nodelist = add_to_nodelist(
        nodelist,mkkern(wd * u))
    end
  end
  -- barcode_top will become the vbox as in the
  -- first solution.
  local barcode_top = node.hpack(nodelist)
  barcode_top = add_to_nodelist(
    barcode_top,mkkern(tex.sp("-1.7mm")))

  -- The following list holds the displayed digits.
  nodelist = nil
  for i,v in ipairs({split_number(str)}) do
    for j=1,string.len(v) do
      nodelist = add_to_nodelist(
        nodelist,mkglyph(string.sub(v,j,j)))
    end
    if i == 1 then
      nodelist = add_to_nodelist(
          nodelist,mkkern(5 * u))
    elseif i == 2 then
      nodelist = add_to_nodelist(
          nodelist,mkkern(4 * u))
```

```
    end
  end
 local barcode_bottom = node.hpack(nodelist)
 -- barcode_top now has three elements: the hbox
 -- from the rules and kerns, the kern of -1.7mm
 -- and the hbox with the digits below the bars.
 barcode_top = add_to_nodelist(
     barcode_top,barcode_bottom)
 local bc = node.vpack(barcode_top)

 -- node.write() puts a vbox into the output.
 node.write(bc)
end
```

The overall structure is exactly the same as in the previous section. The main difference is the use of the helper functions `mkrule()`, `mkkern()` and `mkglyph()` to create rules, kerns and glyphs and the call to `add_to_nodelist()`. The constructed node list is written to the PDF with the Lua call `node.write()`.

```
function add_to_nodelist( head,entry )
  if head then
    -- Add the entry to the end of the nodelist
    -- and adjust prev/next pointers.
    local tail = node.tail(head)
    tail.next = entry
    entry.prev = tail
  else
    -- No nodelist yet, so just return the new entry.
    head = entry
  end
  return head
end
```

If the node list exists, the new entry is appended to the last node of that list. We could get to the end of the list by following successive pointers until we reach the one with the "empty" pointer *nil*, but we use the LuaTEX function `node.tail()` instead. Then we adjust the next and prev pointers of the tail and the new entry and return the head of the node list.

```
function mkrule( wd,ht,dp )
  local r = node.new("rule")
  r.width = wd
  r.height = ht
  r.depth = dp
  return r
end

function mkkern( wd )
  local k = node.new("kern")
  k.kern = wd
  return k
end

function mkglyph( char )
  local g = node.new("glyph")
  g.char = string.byte(char)
  g.font = font.current()
  g.lang = tex.language
  return g
end
```

These three functions don't need much explanation. They generate the nodes of the requested types. It might surprise at first glance that the glyph node needs a language and a font entry, because in ordinary TEX we usually don't care about this. But remember that the nodes are the low-level data structures created when all of TEX's input is already processed, except for the hyphenation and justification of the paragraph.

As a final note on the source, the Lua file described here can be downloaded from `https://gist.github.com/1513746`.

5 Conclusion

There are two ways to pass typesetting information from Lua to TEX: first, with a collection of `tex.sprint()` calls, and second, with a set of nodes.

Once you are in the Lua world, it feels wrong to pass information to TEX with `tex.sprint()` calls. You still have to deal with category codes, with grouping and with all the headaches that character escaping brings.

In the procedural world of Lua, the *right* way to do typesetting is to construct the input with low-level data structures and helper functions and let TEX's algorithms do the rest. Once you start thinking in terms of nodes and node lists, you can focus on arranging items on the page and not let TEX's input language get in your way.

These days, TEX's input language seems anachronistic to many people, while procedural languages like Lua are familiar. TEX's algorithms are still unsurpassed, so when you combine Lua's power with TEX's typesetting capabilities, a whole new generation of applications become possible.

References

[1] Patrick Gundlach. TEX without TEX. `http://wiki.luatex.org/index.php/TeX_without_TeX`, 2011.

[2] Taco Hoekwater. LuaTEX reference manual. `http://mirror.ctan.org/systems/luatex/base/manual`, 2011.

[3] Herbert Voß. The current state of the PSTricks project. *TUGboat*, 31(1), 2010.

◇ Patrick Gundlach
Eisenacher Straße 101
10781 Berlin
Germany
patrick (at) gundla dot ch

OpenType fonts in LuaTEX

Paul Isambert

1 Introduction

As is well-known, LuaTEX can handle standard font formats, notably including OpenType. That's a welcome development because modern font designs use those formats almost exclusively, and whatever the merits of METAFONT, for modern typographic software to stick to it would be suicidal. Lesser known perhaps is that, unlike XETEX, which opened the way, LuaTEX is completely unable to load such a font if you don't feed it a non-negligible amount of code beforehand. Otherwise it only understands your old TFMs (it actually embeds Type 1 fonts in PDF documents, a behavior inherited from PDFTEX, but only because a mapping exists between the TFM and Type 1 fonts; the latter can't be read directly). The reason is not that LuaTEX isn't so capable after all and you have to rely on some work-around; rather, LuaTEX is consistent with its philosophy (as I see it): it provides tools, not solutions. So you have to do most of the work to make it understand OpenType fonts, and that's no simple work, but in the process you gain freedom.

In this paper I'll try to describe such code. I won't give an entire implementation, and in many places I'll just go with "This or that should be done", because as already mentioned it would be extremely long (and tedious). ConTEXt's fontloader, available for plain TEX and LATEX as `luaotfload`, is more than 10,000 lines long, and my own code, which doesn't even try to address non-Latin typography, is 2,000 lines. In other words, I'll give a map of the area to the reader, but nothing can replace the actual exploration.

Also, this paper has limitations: first, all my examples will use the Latin alphabet, even though some features would have been better illustrated with other scripts; I apologize to users of other writing systems, but I thought it better not to pretend I was competent in them. This extends to maths, so OpenType maths aren't covered at all; [2,7] should help the interested reader. The omission of maths is even more significant than for non-Latin writing systems, since the latter at least rely on the mechanisms described here, whereas OpenType maths are an entirely distinct area.

Second, I will have nothing to say about AAT fonts. I have never used them, let alone figured out how they work, and anyway that would have pushed the length of this paper beyond reasonable limits. I hereby invite the courageous reader to tackle the issue and write a companion paper.

Third, this article is an introduction to how LuaTEX sees OpenType fonts, not to the OpenType format itself. Of course, the two are closely related, and after reading this, the Microsoft documentation [5] or a general introduction like [1] will look familiar; but there are significant differences too.

Fourth, this explains how LuaTEX sees such fonts *at the present time*. That is bound to evolve, and some of what is said here will become obsolete. Nonetheless, the knowledge gained in OpenType fonts themselves will not, I hope, be wasted.

Fifth, although I hope the reader will feel comfortable with the subject after reading this paper — or, at least, the reader will feel s/he could be comfortable with the subject after reading the paper thoroughly a few times — nothing replaces experimenting with fonts directly. Fortunately, LuaTEX's fontloader is based on George Williams's FontForge, so there exists a GUI counterpart to all the Lua tables we will explore (modulo the previous point). I strongly recommend playing around in FontForge, tweaking fonts to see what changes in LuaTEX, etc.; also reading the FontForge documentation [8].

I will use a single font as illustration. It is a modified version of Philipp H. Poll's Linux Libertine (italic) [6], renamed Test Libertine. The file is available from `tug.org/TUGboat/tb33-1`, along with a `README` listing all the changes made to the original font. In the course of the article, when I write that "Libertine has such and such feature", I always mean the modified Libertine: the feature at stake may not be present in the original. None of the modifications improves Linux Libertine in any way, nor do they have much value by themselves, either typographically or technically; I've added each and every one of them with a single purpose in mind: to serve as an illustration for this paper.

2 The `define_font` callback

Loading an OpenType font requires that we intercept the user's font request and replace LuaTEX's default behavior with code of our own. As usual this is done thanks to a callback: `define_font`. It is called whenever the `\font` control sequence is used,

Author's note: I am not a member of the LuaTEX team, nor should this article be considered an official introduction to fonts in LuaTEX. However, Taco Hoekwater has answered myriad questions, both for this paper and when I was investigating fonts for myself, and many details here would have been obscure or altogether missing if not for his help. All remaining inaccuracies and outright errors are of course mine.

with three arguments: `name`, `size` and `id`. The first two arguments have direct equivalents in the syntax of `\font`:

`\font\myfont=`⟨*name*⟩ `at|scaled` ⟨*size*⟩

where ⟨*name*⟩ is anything between braces or double quotes or a string of non-blank characters. The ⟨*size*⟩ part is passed to the function registered in the callback as follows: if positive, it represents `at` ⟨*size*⟩ in scaled points, e.g. `at 10pt` becomes `655360`. If negative, it represents `scaled` ⟨*size*⟩, e.g. `scaled 500` becomes `-500`. What if no `at` or `scaled` information is given? Then ⟨*size*⟩ is set to `-1000`, as if `scaled 1000` had been specified, which is equivalent to no scaling at all, since TEX scales fonts by a factor of one-thousandth of the given value.

The `id` argument is the numerical representation of the font. Indeed, TEX internally records fonts as numbers, not names. This can be seen in LuaTEX when you query a glyph node's `font` field: it returns a number. For us, it will be useful when applying OpenType features: characters whose `font` is one we've loaded ourselves and containing special features will require our attention. But we'll see that in due time.

The function registered in `define_font` must return a table of the type which LuaTEX understands.[1] To do so, it reads the appropriate font file. I've said earlier that LuaTEX can't load OpenType fonts. That is not exactly true: it can load such a font (otherwise the remark above about its fontloader being based on FontForge wouldn't make sense), it just doesn't know what to do with it. LuaTEX reads an OpenType font file, creates a Lua table with it, and your job is to turn it into another table that the engine can use. In essence, given a table as described in section 4.4.5 of the LuaTEX reference manual [4], you have to produce a table as described in chapter 7 of the same document. Most of this paper deals with such a transformation. (If you truly have nothing else to do, you can also produce the latter table directly from the font file, not relying on LuaTEX's interpretation.)

We'll name our function the same as the callback itself: `define_font`. That means that ultimately something like the following must occur, after our function is properly defined:

```
callback.register("define_font", define_font)
```

Also, since managing fonts is a matter of (sometimes quite complex) tables, it'll help to have a function that prints the contents of a table in a readable fashion. Here it is (all Lua code in this paper is supposed to be written in a `.lua` file, not in `\directlua`, unless catcodes are properly set):

```
local rep, write = string.rep, texio.write_nl
function ExploreTable (tab, offset)
  offset = offset or ""
  for k, v in pairs(tab) do
    local newoffset = offset .. "  "
    k = offset .. k .. " = "
    if type(v) == "table" then
      write(k .. "{")
      ExploreTable(v, newoffset)
      write(newoffset .. "}")
    else
      write(k .. tostring(v))
    end
  end
end
```

This function browses entries in no particular order; however, in this paper, I will often rearrange its output (sometimes adding commas) to impose some organization. Thus the reader shouldn't worry if what s/he gets at home looks slightly different.[2]

3 From names to files

TEX traditionally loads fonts by reference to the filename, e.g.:

```
\font\tenrm=cmr10
```

loads the font contained in `cmr10.tfm`. However, XETEX has made popular another way of referring to fonts, namely by their internal names:

```
\font\myfont="Linux Libertine O /I: +smcp"
```

This convenient syntax has been taken over in ConTEXt, thus also in `luaotfload`. The part before the colon denotes a font proper, i.e. a font file: the file containing the italic font of the Linux Libertine O family[3] (i.e. `LinLibertine_RI.otf`, or `fxlri.otf` in TEX Live). After the colon are the tags denoting features to be applied to that font. With such a font call, our function will be executed as:

```
define_font("Linux Libertine O /I: +smcp",
            -1000, 51)
```

(provided we are defining the 51st font, which is the case if this is the first font call after loading plain TEX). Now, if we ask LuaTEX to load a font named "`Linux Libertine O /I: +smcp`", or even, since we're not so naïve, "`Linux Libertine O /I`", it will never find it. Instead, it must be given a filename and nothing else, so you have to link names to files. To do so, we have to open all the font

[1] Actually, a number can also be returned, which will be interpreted as the `id` of another, already defined font. This possibility won't interest us much, of course.

[2] Also, although the resulting table *looks* like a Lua table, it is not (and thus shouldn't be reused as such in Lua code). I leave it as an exercise to the reader to list the differences.

[3] The *O* in the family name is for OpenType.

files available to LuaTEX, as specified by the return value of `kpse.show_path("opentype fonts")`, and check their names. That takes quite a while, so it's not something we want to do on each compilation. Thus we build a database as a Lua table to match names with files, e.g. (Biolinum is the sans-serif companion to Libertine):

```
database = { ...
  ["Linux Libertine O"] = {
    Regular = "LinLibertine_R.otf",
    Italic  = "LinLibertine_RI.otf",
    ... },
  ["Linux Biolinum O"] = {
    Regular = "LinBiolinum_R.otf",
    Italic  = "LinBiolinum_RI.otf",
    ... },
  ... }
```

The first time the fontloader is used, it will create the database and write it to an external file. After that, linking names to files will be fast. Given e.g. `Linux Libertine O /I`, it can easily find that `LinLibertine_RI.otf` is required, since by convention `/I` means italics. To emphasize, this is just a *convention*, and you could decide to use `/Italic` instead, or even to allow the user to specify anything between `I` and `Italic` (then you check whether it is the prefix of a tag). The latter solution is convenient for fonts which have both Bold and Black variants (the `/B` tag is ambiguous). Also, nothing prevents the database from being written as something more readable and/or modifiable. I use a file with a very simple syntax that I can customize by hand, so that I can for instance simplify family names: "Linux Libertine O" becomes simply "Libertine", "Pro" is removed in the names of many Adobe fonts ... or I can lump unrelated files into a single family. An imaginary but familiar example would be:

```
Centaur:
  Regular = "Centaur.otf"
  Italic  = "Arrighi.otf"
```

Finally, the database can be used to retrieve TFM fonts by name too.

Fine: we know what the database should or could look like, but how are we supposed to create it? The answer has already been hinted at above: we browse all directories where OpenType fonts are supposed to be, open all files, retrieve information and store it in a table. In short, we do the following:

```
local open = fontloader.open
database = {}
local storeinfo
function explorefonts (dir)
  if not lfs.isdir(dir) then
    return
  end
  for name in lfs.dir(dir) do
    if name ~= "." and name ~= ".." then
      local file = dir .. "/" .. name
      local attr = lfs.attributes(file)
      if attr then
        if attr.mode == "file" and
         name:lower():match("%.[to]tf$") then
          storeinfo(file, name)
        elseif attr.mode == "directory" then
          explorefonts(file)
        end
      end
    end
  end
end

function storeinfo (file, name)
  local i = open(file)
  if not i then
    return
  end
  local fam, mod
  for _, lang in ipairs(i.names) do
    local n = lang.names
    fam = n.preffamilyname or n.family
    if fam then
      mod = n.prefmodifiers or n.subfamily
      database[fam] = database[fam] or {}
      database[fam][mod or "default"] = name
      break
    end
  end
end
```

Before studying what this code does, a few words on programming details. I won't be necessarily consistent with regard to `local` variables; in production code, most declarations would be local (including `define_font`, since there's no problem with registering a local function in a callback). Here, however, variables will be local if, in a given code snippet, they are used in other functions only; if they are toplevel with respect to the current snippet, they will be defined globally. Hence `storeinfo` is local because we won't be using it directly, whereas `explorefonts` is global, as is the `database` table. With respect to the locality of `storeinfo`, the reader might notice that it is declared before `explorefonts` (so it is available to the latter function) but defined (without `local`) after it (so we read its definition only when we know where it is supposed to work); this is a convenient way to organize code in this situation (in other situations, e.g. with two `local` functions calling each other, it is the *only* way). Finally, both functions use:

```
if ⟨something⟩ then
  return
end
⟨code⟩
```

instead of

```
if ⟨something⟩ then
  return
else
  ⟨code⟩
end
```

Both forms achieve the same result; in general, I use the second style, because I find it clearer, but in some cases, as in here with *TUGboat*'s narrow columns, the first style saves precious space (and avoids a lonely `end` whose role might be obscure).

Back to the code itself. We can use it like this:

```
local fontpath = kpse.show_path("opentype fonts")
fontpath = fontpath:match(":")
           and fontpath:explode(":")
           or fontpath:explode(";")
for _, dir in ipairs(fontpath) do
  dir = dir:gsub("^!!", "")
  explorefonts(dir)
end
```

We ask `kpathsea` where OpenType fonts live, and it returns a string of paths separated by colons or semi-colons (the latter case on Windows); we remove exclamation marks that might prefix a path for reasons that won't concern us here, and launch our function `explorefonts`: using the `lfs` (LuaFileSystem) library, we browse the contents of a directory (making sure it *is* a directory, because `kpathsea` stores *possible* paths); for each element, if it is a file and it has the proper `otf` or `ttf` extension, we pass it to our `storeinfo`; if it is a directory, we browse it recursively. The reader may be surprised that we take `ttf` files into account; but there are two breeds of OpenType fonts: OpenType C[ompact] F[ont] F[ormat] files have the `otf` extension, whereas OpenType TrueType files have the `ttf` extension. However, the `kpse` library sees the latter as TrueType fonts (because of the extension), and when searching for a font in that format we need to specify `"truetype fonts"` as the type.

With `storeinfo` comes our real taste of OpenType fonts with LuaTeX. We use the `fontloader.open` function to import a file into a readable format (albeit not terribly readable, as we'll learn later), and if it worked (this might not be the case, e.g. if permission is denied) we retrieve the information we need. Before turning to that, though, I should mention that there exists `fontloader.info`, a function that extracts exactly the information we'll need, and which is much faster than `fontloader.open`. However, `fontloader.info` also regularly gets things wrong, not due to a defect in LuaTeX but because fonts often are badly organized. For instance, in the table returned by `fontloader.info`, the field `familyname` for Robert Slimbach's Minion Pro bold is set to `Minion Pro`, but it is `Minion Pro SmBd` for the semibold version (and the reverse holds for Adobe Caslon, Carol Twombly's adaptation of William Caslon's famous design), as if they belonged to different families. Also, the only information about the 'italic-ness' of a font is the `italicangle` field, which might have a non-zero value even if a font is not italic in the sense of being related to a roman alternative (see for instance calligraphic fonts).[4]

Anyway, we have our font loaded in the table-like (technically: `userdata`) variable `i`, which has a field `names`, an array of subtables with information about the font in different languages, as follows:[5]

```
names = {
  1 = {
    lang  = English (US)
    names = {
      family = Test Libertine
      subfamily = Italic
      fullname = Test Libertine Italic
      postscriptname = TestLibertine
      uniqueid = FontForge 2.0 : Test ...
      version = Version 5.1.1
      designer = Philipp H. Poll
      manufacturer = Philipp H. Poll
      designerurl = http://www. ...
      vendorurl = http://www.tug.org
      licenseurl = http://www.fsf.org/...
      copyright = Test Libertine by ...
      license = GPL ...  } }
  2 = { lang = German German
        names = { subfamily = Kursiv } }
  ...
  }
```

There is often only one such subtable (the first shown here), but sometimes there are several, containing some information in various languages.

The information we're looking for is `family` and `subfamily` or, if they exist, `preffamilyname` and `prefmodifiers`; they correspond to the font name and slash-prefixed tags in a `\font` call with XeTeX syntax. Some modifiers should be filtered

[4] Actually, it should be possible to do a meaningful analysis of the table returned by `fontloader.info`, but it is just simpler to use `fontloader.open`. It is indeed much slower, but then the database isn't built on every compilation, so it's not so bad if it takes a little time.

Still, there is one case where we *need* `fontloader.info`: TrueType Collection files. This format puts several fonts in one file (with the `ttc` extension, so not taken into account here); `fontloader.info` returns an array of tables similar to the single one it returns for other files; to load a particular font in a collection with `fontloader.open`, we must pass a second argument to the function, the `fontname` found in one of the tables returned by `fontloader.info`.

[5] This shows `names` for Test Libertine, because that's the font we'll be looking at in detail, but since I've modified only one font it doesn't really belong to a family, and the original Linux Libertine is used as an example elsewhere in this discussion of the font database.

out, namely `Regular`, `Book` and others, because they denote the "default" font, i.e. roman with normal weight, and we don't want to have to specify such a font as:

```
\font\myfont="Linux Libertine O /Regular"
```

Also, a modifier isn't always as expected. For instance, William A. Dwiggins's Electra in italics is called `Cursive`; you probably want to normalize that to `Italic`, so it can be called like other italic fonts.

Now, building the database is a simple matter: it will be a table with family names as entries whose values are subtables with ⟨*modifier(s)*⟩ = ⟨*filename*⟩ subentries. The database is cached, and when the user specifies:

```
\font\myfont="Linux Libertine O /I : +smcp"
```

we retrieve the value of the `Italic` subentry of the "`Linux Libertine O`" entry and, lo, we are magically directed to the proper file! Admittedly, this requires a bit of string manipulation (left as an exercise to the reader), and we don't know what to do with `+smcp` yet, and in fact we don't even know what to do with the font file itself, but let's shout triumphantly anyway, it's good for morale, and we'll need some because this paper won't get any easier.

4 Basic entries in a font table

The `fontloader.open` function loads a font, but it's not usable by itself; the result should be turned into a table with `fontloader.to_table`, as follows. (The `close` operation simply discards the userdata from which the table is extracted and requires no further comment.)

```
local f = fontloader.open
        ("/your/font/dir/TestLibertine.otf")
fonttable = fontloader.to_table(f)
fontloader.close(f)
```

We shall turn this table into another, as said before. However, all those operations take time, so we'll want to perform them as seldom as possible. That is why the font table should be cached: once a font has been analyzed, relevant information is stored in a file that future compilations will retrieve instead of starting from scratch again. However, the result of some operations can't be cached: e.g. those related to dimensions (sizes of glyphs, etc.) must be performed anew each time the font is loaded, because they depend on the size at which the font is loaded; in the cache file, only "abstract" dimensions are stored, with an arbitrary unit, and they must be converted to fit the real size. Unfortunately, setting kerning pairs (i.e. the — typically negative — amount of space added between pairs of glyphs that don't look good when set next to each other) is one of those operations, and kerning pairs come by the thousands in some fonts (e.g. Minion Pro). Another operation that obviously can't be done in advance and cached is applying features.[6]

Another important remark about size. With TFM fonts, when you call e.g.:

```
\font\myfont=cmr10
```

you're requesting a font with a given size (here 10pt), because that information is part of the font file. But at what size should our Libertine font be loaded? Some fonts have a `design_size` field (expressed in tenths of a point — PostScript point, big point to TEX; we shall ignore that subtlety here); e.g. with `fonttable` above, `fonttable.design_size` returns `110`, meaning 11pt. So we could load the font at 11pt; the problem is that the design size may vary from font to font, so by default you'll be loading Libertine at 11pt and Electra at 12pt; worse still, some fonts don't have a specified design size (the field returns `0`). So it's better to have a default size at which a font will be loaded regardless of its design size; of course the problem vanishes if an `at` ⟨*size*⟩ clause is used in the font call.[7]

Let's get back to loading our font. The operations described in this paper assume we're reading from the original font file, and that nothing is cached (or at best what is cached is the table returned by `fontloader.open/to_table`, i.e. the original font file translated to Lua, so to speak). So we have our original table, which I'll call `fonttable`, and we

[6] One could cache a font at a given size, e.g. 10pt, so that at least when loaded at that size (preferably the most often used, of course), the operations on dimensions are already done. Another option is to cache fully specified fonts, i.e. with size and features applied, for a given job or set of jobs, so that all compilations but the first are faster (under the assumption that the user doesn't change fonts or font specifications on each compilation, of course). Those cache files can then be deleted once the job is done, like auxiliary files in general. Of course the features mentioned here are only those that LuaTEX can handle by itself, as will become clear later.

[7] Actually, an intelligent fontloader, unless instructed otherwise, will try to return the font that best fits the at-size; in many cases there will be one font only that matches the font call, whatever the size; in other cases, though, there will be several, and the right one should be chosen. That happens when a font is drawn at different sizes, as with many Adobe fonts and Latin Modern (by Bogusław Jackowski and Janusz M. Nowacki — and Donald E. Knuth). Of course the font database should reflect the fact that those fonts vary only with respect to size (which shouldn't be thought of as a modifier on a par with those we've been dealing with), something that the database created above didn't do. I won't pursue this matter here, except to mention that not only should `design_size` be taken into account, but also `design_range_bottom` and `design_range_top`, which specifies the (exclusive) lower and (inclusive) upper bounds of the range of sizes for which the font is optimal.

must return another, which I'll call `metrics`. A few fields can be readily set:

```
metrics = {
  name       = fonttable.fontname,
  fullname   = fonttable.fontname .. ⟨id⟩,
  psname     = fonttable.fontname,
  type       = "real",
  filename   = ⟨filename⟩,
  format     = ⟨fonttype⟩,
  embedding  = "subset",
  size       = ⟨size⟩,
  designsize = fonttable.design_size*6553.6
  }
```

First we specify some names: the `name` field is used internally by LuaTEX, e.g. in error messages; `fullname` is suffixed with ⟨*id*⟩ (the third argument to `define_font`) because, in rare cases, fonts with identical names (extracted from the same font file but with different features, e.g. with and without small caps) can cause problems: indeed, if two fonts are sufficiently similar, LuaTEX will merge them in the PDF output; adding ⟨*id*⟩ avoids the merging; as for `psname`, it is relevant to the PDF file. The `type` distinguishes `real` from `virtual` fonts as TEX has always done, but I won't address virtual fonts here.

LuaTEX uses info in a font to know what glyph to place where; but the PDF file must contain the file to render the glyphs, and that's the meaning of the following entries: the `filename` field must contain the full path to the original font file, i.e. `/your/font/dir/TestLibertine.otf` in our example, so that LuaTEX can embed it. The `format` must be one of `type1`, `type2`, `truetype` and `opentype` (the latter in our case; note that TrueType-based OpenType fonts, i.e. with the `ttf` extension, should use the `truetype` format). Finally, the `embedding` field specifies what the PDF file should contain of the original font file: if the value is `no`, the font won't be embedded at all, and the PDF viewer will try to find it on the disk[8] or, failing that, it will use a default font (but the glyphs will be placed according to what LuaTEX will have read from the original font, so the result, quite obviously, will be a mess); the value `subset` means that only those glyphs that are used in the document are described in the PDF file; finally, `full` means that the PDF document contains the entire font file. What to embed is a matter of size and license; commercial font vendors generally allow subset embedding, which is the best solution anyway, but strictly speaking that should be checked beforehand. (E.g., via the `license` and/or `licenseurl` fields in the `names` table as shown above, and more precisely, the field `fonttable.pfminfo.fstype`, corresponding to the `fsType` entry of the `OS/2` table in the original font file; that is a 16-bit number each bit of which is a boolean. Suffice it to say for us that subset embedding is perfectly ok if the number is 0 or 8.)

We now turn to the matter of size. The value of ⟨*size*⟩ depends on the font call and the contents of the at-size clause (recall our earlier discussion); if the called font contained an explicit `at` ⟨*size*⟩, then that is the value of the variable; otherwise, it contained a `scaled` ⟨*factor*⟩ clause, perhaps implicit (i.e. no clause was given, which is equivalent to `scaled 1000`); then `-`⟨*factor*⟩ was passed to `define_font`, and we have to agree on a default size to do the scaling. Suppose that default is d and we denote ⟨*factor*⟩ with f; then ⟨*size*⟩ is $-\frac{f\times d}{1000}$. (This obviously yields d if the font was called without `scaled`.) LuaTEX internally treats all dimensions in scaled points: we should never try to pass it `10pt`, and `655360` should be used instead (since there are 65,536 scaled points per TEX point). The `tex.sp` function can be used: when passed a dimension (as a string), it returns its value in scaled points. In short, a snippet from `define_font` would be (given a default size of 10pt):

```
function define_font (name, size, id)
  ...
  if size < 0 then
    size = size * tex.sp("10pt") / -1000
  end
  ...
end
```

(The observant reader can infer from this code and the discussion above that `size`, if positive, has already been converted to scaled points when passed to the function.)

So, the `size` field is set to the result of that tediously explained computation. It is used not by LuaTEX, which relies on the glyphs' sizes themselves, but by PDF viewers, which will draw glyphs at that size. A mismatch between this value and the real size at which the font was loaded will thus result in a mismatch between the drawn glyphs and the space they occupy (letterspacing can be bluntly implemented this way). As for the `designsize` value, it is used by LuaTEX when reporting information with `\fontname`, for instance, in which case `at` ⟨*size*⟩ will be mentioned if it differs from `designsize` (if the original `fonttable.design_size` is 0, i.e. unspecified, it's better just to set `designsize` to `size`).

Now that we have learned about ⟨*size*⟩ we can turn to another important field of `metrics`: namely `parameters`, a table with seven entries. This num-

[8] For this, `psname` is crucial; if `psname` is missing, LuaTEX will use `fullname` which, when suffixed with ⟨*id*⟩ as is the case here, will not be a name that is findable on disk.

ber might remind the seasoned TEX user of something with which s/he is familiar: the `\fontdimen` primitive. Indeed, that's where those dimensions are set: `\fontdimen`n is the entry at index n in `parameters`. However, parameters 1–7 have been given friendlier names: LuaTEX will use entries with those names as keys if they exist, and fall back to the numbered entries otherwise. Here's a brief description of those parameters (math fonts have entries at index 8 and higher, but we won't investigate those here); more complete descriptions can be found in your favorite reference:

1. `slant` Slant per point, for accent positioning.
2. `space` Interword space.
3. `space_stretch` Interword stretch, i.e. the space that can be added to the interword space when a line is justified by stretching.
4. `space_shrink` Interword shrink, i.e. the space that can be subtracted from the interword space when a line is justified by shrinking.
5. `x_height` Value of the unit `ex`.
6. `quad` Value of the unit `em`.
7. `extra_space` Space added when `\spacefactor` ≥ 2000.

A few of these parameters can be set from information found in OpenType fonts: `x_height` can be read in `fonttable.pfminfo.os2_xheight` or derived from the height of the letter x in the font (a better solution), but that requires a conversion we'll turn to presently; thus the value given below for that parameter is arbitrary. The interword `space` should be the width of the space character, but again we don't know how to retrieve that yet. Finally, `slant` can be derived from `fonttable.italicangle` if the latter is given. Here I have specified interword space and associates as in Computer Modern (10pt):[9]

```
local T, R = math.tan, math.rad
metrics.parameters = {
  slant = 65536/T(R(90 + fonttable.italicangle)),
  space         = ⟨size⟩ / 3,
  space_stretch = ⟨size⟩ / 6,
  space_shrink  = ⟨size⟩ / 9,
  x_height      = 0.4 * ⟨size⟩,
  quad          = ⟨size⟩,
  extra_space   = ⟨size⟩ / 9
  }
```

Those fields could also get their values from the font call; remember that, in keeping with the XETEX syntax, whatever comes after the colon (if any) will be interpreted as features to be applied to the font (e.g. ligatures, kerning, etc.). We could also allow additional information to be given, so the user could ask for something like this:

```
\font\myfont="Test Libertine /I: stretch=.2;..."
```

to mean that `space_stretch` should be set to a fifth of the loading size (among other features); of course, such a parameter could also be set by hand with `\fontdimen`, but this is a nicer interface.

Before extracting the marrow from OpenType fonts, I shall say that there are many other fields in `fonttable` that we won't explore here because they're not crucial, even though we could make use of them. Interesting information on the font can for instance be found in the `fonttable.pfminfo` table, which lumps together fields from (mostly) the `hhea` and `OS/2` table of the original font file. Similarly, some other fields in `metrics` could be set that we won't consider here.

5 Glyphs, at last!

Now that we've gone through all the preliminary steps we can turn to the crux of the matter: glyphs. In `fonttable`, there is a `glyphs` subtable which contains them all, and we shall use them to populate the `characters` entry in `metrics`.[10] However, entries in the `fonttable.glyphs` table are arbitrary, whereas `metrics.characters` indices are Unicode codepoints; for instance, *a* in Libertine (italic or not) is at index 66 in `fonttable.glyphs`, even though its codepoint is 97, which is also the index where it must appear in `metrics.characters` (unless we're implementing substitutions, but we won't be doing that now).

So, we need to know: a) the characters the font contains and b) the glyph number of each character.

[9] The computation for `slant` is not complicated, but it might not be very readable as expressed here. So here are the steps:

1. `slant` is the horizontal displacement for one point of vertical displacement. Hence it can be expressed as $1 / \tan\alpha$, where α is the angle between the x-axis and the font's axis.
2. `fonttable.italicangle` measures the angle between the y-axis and the font's axis (which is why it is negative for fonts leaning on the right, as most slanted fonts do); α is thus: $-(90-$`fonttable.italicangle`$)$, i.e. $90+$`fonttable.italicangle`.
3. α is expressed here in degrees, but Lua's `math.tan` function expects radians, hence the use of `math.rad` to do the conversion.
4. The result is expressed in points; we multiply it by $65,536$ to convert it to scaled points.

For fonts with proper information for diacritic positioning, `slant` is useless; we'll use OpenType features instead. But setting it correctly does no harm.

[10] My use of the words *character* and *glyph* doesn't reflect the common distinction between an element of a writing system and its representation (so that the character *a* can be represented by various glyphs, e.g. from different fonts); instead I will use *glyph* for an element of `fonttable.glyphs` and *character* for an element of `metrics.characters`.

Fortunately, we have `fonttable.map.map`: it is an array with Unicode codepoints as indices and glyph numbers as values, e.g. `97 = 66`, meaning that the character with Unicode codepoint 97 has its glyph in `fonttable.glyphs[66]`. Let's give a shorter name to this table and use it to look at the glyph *f*:

```
map = fonttable.map.map
ExploreTable(fonttable.glyphs[map[102]])
```

And the result is (somewhat shortened):

```
name    = f
unicode = 102
class   = base
width   = 314
boundingbox = { 1 = -78
                2 = -238
                3 = 523
                4 = 698 }
kerns   = { ... }
lookups = { ... }
anchors = { ... }
```

I have elided the `kerns`, `lookups` and `anchors` table since we aren't able to do much with them for the time being. The first two fields are quite obvious, I suppose; `name` will be of use later, exactly when we'll examine the contents of `kerns` and `lookups`. We'll ignore the `class` field until we start discussing lookups. The `width` field is, unsurprisingly, the width of the glyph. The values in the `boundingbox` table are the position of the glyph's extrema: `1` and `3` are the minimum and maximum x-values respectively, while `2` and `4` are the minimum and maximum y-values; the former are expressed relative to the y-axis (i.e. the left side of the glyph's bounding box proper, where $x = 0$), and the latter relative to the x-axis (the glyph's baseline). Thus `boundingbox[4]` is its height, `-boundingbox[2]`, provided `boundingbox[2]` is negative, i.e. the lowest point is below the baseline, is the glyph's depth, but the glyph's width is `width`, and nothing else! Indeed, a glyph as drawn may be larger than its declared width, and it may extend outside its bounding box, and that's perfectly normal. Figure 1 illustrates that with the glyph we're investigating (the image is from FontForge): the glyph's width is the area between the two vertical lines; the extenders aren't contained between those lines, which means that, for instance, an *i* before the *f* will stand above the *f*'s tail, while an *o* after will stand below its arm: *ifo*. That's welcome behavior, otherwise spurious gaps would occur between letters.

Now, the only fields LuaTEX *requires* for a character are `width`, `height`, `depth` and `index`, the latter being the glyph's index in `fonttable.glyphs`. In fact, LuaTEX is directly interested in the first three fields only: they are the basic data it requires

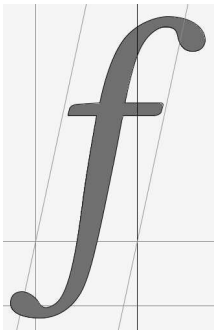


Fig. 1: *f* in Libertine Italic

to do its job properly. On the other hand, `index` is used to denote the glyph in the PDF file.[11] So, that's it, we can transfer glyphs in `fonttable` to characters in `metrics`! Oh, no, we can't: we know *f*'s width is 271, but 271 *what*? Scaled points? No, that would be too easy.[12] The unit in which dimensions are expressed in font files is a relative unit, which makes sense since the font may be loaded at whatever size. The value of that unit is recorded in `fonttable.units_per_em`, generally 1,000 or 2,048, but the real value is of little importance: what counts is that, given ⟨*size*⟩ as computed above, we are able to derive a (this time absolute) unit for interpreting glyph dimensions. Since by definition ⟨*size*⟩ is one em, then the value of the unit is obviously $\frac{s}{u}$ with s = ⟨*size*⟩ and u = `fonttable.units_per_em`. Let's record it in a variable:

`unit =` ⟨*size*⟩ `/ fonttable.units_per_em`

and here we go, let's translate `fonttable`'s glyphs into `metrics`'s characters (recall how `map` was defined above):

```
for ch, idx in pairs(map) do
  local glyph = fonttable.glyphs[idx]
  metrics.characters[ch] = {
    index  = idx,
    width  = glyph.width * unit,
    height = glyph.boundingbox[4] * unit,
    depth  = glyph.boundingbox[2] < 0 and
             (-glyph.boundingbox[2] * unit) or 0 }
end
```

This time, that's it, we're done! The `define_font` function can return `metrics` and LuaTEX will be able to use it. No kidding: extracting the glyphs is child's play by comparison.

[11] Indeed, *TeX* will be written ⟨*003500460039*⟩ in the PDF file, instructing the viewer to fetch glyphs at position `0x35`, `0x46` and `0x39` in the current font (assuming the current font is Libertine). Actually, things are a bit more complicated than that, but we definitely don't want to dwell on PDF.

[12] Not to mention that a glyph with a width of 271 scaled points would be a billboard for atoms but not exactly for us.

6 Some easy-to-implement niceties

Okay, well, we're done, but let's face it: our font isn't that exciting. If this is all we can do with OpenType, that's rather disappointing. The font isn't even being respected, since kerning information has been ignored.

Of course, we will be doing much, much more. But we'll postpone that as long as there are simpler areas to investigate. For instance, the letter *f* wasn't randomly selected to illustrate the previous section: I chose it because it is the letter that requires italic correction *par excellence*. To wit: "*arf*' must be uttered by an uneducated dog, whereas "*arf*" is from a dog with manners. As the reader certainly knows, it's the difference between

```
``{\it arf}''  and  ``{\it arf\/}''
```

where `\/` denotes italic correction, a small amount of space to be added after the letter. The problem is that italic correction was born with TFM format and didn't prosper. In other words, there is no such thing in OpenType fonts. However, we can mimic it: we'll define italic correction as the difference between a glyph's rightmost point and its width. Given that LuaTEX stores the italic correction in a character table's `italic` field, we can thus enhance the `char` table defined above as follows (where `glyph` is as before):

```
char.italic = (glyph.boundingbox[3] - glyph.width)
              * unit
```

The results might be more or less felicitous, since italic correction was meant to be specified for each glyph by the designer, not automatically computed, but I find this much better than no italic correction at all.[13]

Other things can be easily implemented, this time properly, because they're just the Lua version of existing operations. For instance, the `extend` field in `metrics` corresponds to the `ExtendFont` keyword in a PDFTEX map file: it lets you stretch or shrink the glyphs. The value ranges between $-5,000$ and $5,000$; glyphs are then horizontally distorted by a thousandth of the given value (so that with $1,000$ the font is untouched); a negative value reverses the glyphs. The glyphs' widths are not actually modified; extension takes place in the PDF. In other words, LuaTEX sees and positions them with their original size, thus proper extension should also modify the glyphs' horizontal dimensions (something not possible in PDFTEX).

The `slant` entry in `metrics`[14] corresponds to the `SlantFont` keyword in a map file and allows creation of artificially obliqued versions of a font (or artificially upright versions of a slanted font). It ranges between $-2,000$ and $2,000$ with 0 meaning no slant (other than the font's native slant, of course). The calculation is as follows: given a value of s for `slant`, the resulting angle between the y-axis and the (modified) font's axis is $-\arctan\frac{s}{1000}$ (under the assumption than the font is originally unslanted; otherwise, add that to the font's angle). For instance, if `slant` is $1,000$, then the font will make an angle of $-45°$ (or $45°$ clockwise) with the y-axis. Again, LuaTEX doesn't take that value into account when positioning glyphs, and artificial slant should be paired with increased italic correction.

Let's stop disfiguring fonts with electronic surgery and turn instead to something definitely useful: character protrusion and font expansion (a.k.a. HZ for Hermann Zapf). Both have TEX interfaces, but we can define them here at once when loading the font: instead of using `\lpcode` and `\rpcode` for protrusion and `\pdffontexpand` and `\efcode` for expansion, we can specify those values in Lua. For each character, we can set the `left_protruding` and `right_protruding` fields, which correspond to `\lpcode` and `\rpcode` respectively (the values are thousandths of ems).[15] As for expansion, the `\pdffontexpand` primitive is reflected as follows in Lua. To the statement

```
\pdffontexpand\myfont
  ⟨stretch⟩ ⟨shrink⟩ ⟨step⟩ [autoexpand]
```

corresponds to the following settings in `metrics`:

```
metrics.stretch = ⟨stretch⟩
metrics.shrink  = ⟨shrink⟩
metrics.step    = ⟨step⟩
metrics.auto_expand = ⟨boolean⟩
```

where specifying `true` for ⟨*boolean*⟩ is equivalent to using the `autoexpand` keyword (which is highly recommended, otherwise you need several versions of the same font). Finally, each character can have a field `expansion_factor` corresponding to `\efcode`.

As noted above, both protrusion and expansion can be set in TEX;[16] however, the Lua way can be

[13] One could also use an external file to store italic corrections for a given set of glyphs from a given font, and retrieve the values and apply them when the font is loaded. This might seem like overkill, but it's no more tedious than checking kerning pairs. Also, LuaTEX is of little use if not for such subtleties.

[14] Not to be confused with the previously-discussed entry of the same name in `metrics.parameters`.

[15] Character protrusion in LuaTEX is still buggy as I write this (with v0.71); negative protrusion isn't properly obeyed.

[16] Also, one must set the parameters `\pdfprotrudechars` and `\pdfadjustspacing` to 1 or 2 if protrusion and expansion are to be performed, or use the Lua formulations `tex.pdfadjustspacing` and `tex.pdfprotrudechars`. Perhaps even set

the basis for a nice interface in the XeTeX-like syntax of the font call, e.g.:

```
\font\myfont="Test Libertine /I:
  expansion = 30 20 5;
  expansion_factor = ⟨whatever⟩"
```

where ⟨*whatever*⟩, as the name indicates, could point to an external file or previously-defined Lua table or, well, whatever contains the individual glyph expansion factors.

Before turning to the implementation of advanced typographic features,[17] we can have an easy first taste of such operations. The studious reader who has followed this paper with a computer next to him/her and experimented with each step will have seen some strange behavior: the line of code

```
``I'm going to be interrupted---''
```

comes out as

``I'm going to be interrupted---"

instead of

"I'm going to be interrupted—"

The reason for this apparent misbehavior is that OpenType fonts don't follow a well-known convention in TeX: namely, that TFM fonts have (single) quotation marks in lieu of the "grave accent" and "apostrophe" characters (hence ‘ instead of `); that two quotation marks in a row are replaced by double quotation marks (hence " instead of ‘‘, but the latter can't be seen here because of the previous point); that two hyphens in a row form an en-dash (-- becomes –); and that an en dash followed by a hyphen turns into an em dash (–- becomes —).[18]

The quotation marks could be easily substituted for the grave accent and apostrophe:

```
metrics.characters[96] = metrics.characters[8216]
metrics.characters[39] = metrics.characters[8217]
```

This is not a good idea, however! It upsets the character/glyph correspondence, and that should never be done lightly (i.e. without keeping track of whatever substitution has been performed).

On the other hand, the "TeX ligatures" can instead be implemented as follows. Characters in `metrics` may have a `ligatures` table which is organized like this:

```
ligatures = {
  [⟨nextidx1⟩] = { char=⟨ligidx1⟩, type=⟨type⟩ },
  [⟨nextidx2⟩] = { char=⟨ligidx2⟩, type=⟨type⟩ },
  ... }
```

where each ⟨*nextidx n*⟩ is the index of a character in `metrics` and ⟨*ligidx n*⟩ is the character being substituted if the current character and ⟨*nextidx n*⟩ are found next to the other. As for ⟨*type*⟩, it is either a number or a string denoting the result of the ligature: either the new character or the new character with one or the other or both of the old characters; ⟨*type*⟩ also specifies where TeX should resume scanning. We will content ourselves with type 0.

So, given that the codepoint of hyphen is 45, en-dash is 8,211 and em-dash is 8,212, we can create our convenient ligatures like this:

```
metrics.characters[45].ligatures = {
  [45] = { char = 8211, type = 0 } }
metrics.characters[8211].ligatures = {
  [45] = { char = 8212, type = 0 } }
```

Although we could start implementing f-ligatures and the like this way also, we'd rather do things properly. So let's turn to the reason why OpenType fonts exist in the first place.

* * *

As we now dive into the core of OpenType fonts, the reader should recall the warning issued in the introduction to this paper: LuaTeX's fontloader will evolve, the organization of `fonttable` will change; thus we should observe one field in particular, not mentioned before: `table_version`. Here I describe a `fonttable` whose version is 0.3.

7 A first look at features

It's simpler to approach features with a real example than with an abstract definition. If we explore again the table for *f* (102) in Libertine,[19] but this time retain the `lookups`, `kerns` and `anchors` tables only, and in each only the first entry, we see:

```
lookups = {
  ss_l_0_s = {
    1 = { type = substitution
          specification = { variant = f.short }
        } }
  ...  }
```

them automatically when loading a font with protrusion and/or expansion enabled.

[17] I mean *advanced* from a software point of view; substitutions and positioning are of course as old as writing.

[18] There are a few other substitutions/ligatures, e.g. `!‘` gives ¡. Those substitutions/ligatures aren't exactly that, if we define "substitution" as the replacement of a glyph with another denoting the same character (here a character is replaced with another character: the grave accent and the left quote aren't the same thing) and "ligature" as the merging of two or more glyphs into a new one for æsthetic reasons (hyphens aren't supposed to happen in a row, and even if they would, there's no typographic convention to the effect that they should be merged into an en-dash). Rather, these are convenient abbreviations.

[19] From now on when exploring a particular character, I'll mention the character itself or its name followed by the Unicode codepoint between parentheses; the resulting table is thus produced with `ExploreTable(fonttable.glyphs[map[⟨codepoint⟩]])`.

```
kerns = {
  1 = { char   = o
        off    = -11
        lookup = { 1 = pp_l_2_g_0,
                   2 = pp_l_2_k_1 } }
  ...  }
anchors = {
  basechar = {
    Anchor-3 = { x = 136, y = 215,
                 lig_index = 0 }
    ... } }
```

This tells us a few things: first, *f* should be replaced by `f.short` (*f* becomes *f*) if lookup `ss_l_0_s` is active. Also, the distance between *f* and *o* should be decreased by 11 units (with units defined as above) if `pp_l_2_g_0` is active.[20] Finally, anchors are specified in case a mark is placed relative to the glyph.

One absolutely crucial point: the `f.short` and `o` glyphs aren't denoted by their Unicode codepoints, nor by their positions in `fonttable.glyphs`, but by their names. When we first looked at *f* we could see that it had a `name` field. This is the case for all glyphs (even if sometimes the name is as uninformative as `uni0358`), and all typographic operations denote glyphs in that way. On the other hand, in `metrics` names are useless and only codepoints matter. That means that we badly need a table linking names to codepoints; here it is:

```
name_to_unicode = {}
for ch, gl in pairs(map) do
  name_to_unicode[fonttable.glyph[gl].name] = ch
end
```

Now, `name_to_unicode["f.short"]` nicely returns 57,568,[21] and the reader who has followed the previous instructions to load Libertine can check that `\char57568` indeed yields *f*. And if we were to rush into the kerning of *f*, we could do:

```
metrics.characters.f.kerns = {
  [name_to_unicode.o] = -11 * units }
```

But let's not do that. Instead, let's conscientiously study how features and lookups work, and first of all, how to know whether a lookup is active or not.

8 Lookups, tags, scripts and languages

A lookup is activated if the right combination of script, language and tag obtains; again, a concrete example will be easier to understand. Substitution lookups are stored in `fonttable.gsub`, which is an array where each entry is a table describing a feature.[22] If we explore entry 19 in Libertine's `gsub`, we get the following:[23]

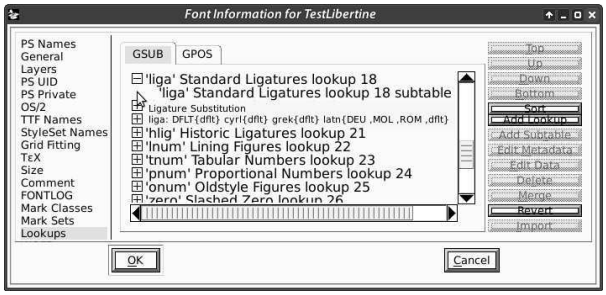


Fig. 2: `ls_l_18` as seen by FontForge.

```
name      = ls_l_18
type      = gsub_ligature
flags     = { }
subtables = { 1 = { name = ls_l_18_s } }
features  = {
 1 = {
  tag = liga
  scripts = {
   1 = { script = DFLT, langs = { 1 = dflt } }
   2 = { script = cyrl, langs = { 1 = dflt } }
   3 = { script = grek, langs = { 1 = dflt } }
   4 = { script = latn,
         langs = { 1 = "DEU ", 2 = "MOL ",
                   3 = "ROM ", 4 = dflt } } } } }
```

(Compare this to Figure 2, which shows the same lookup as seen by FontForge.) Let's focus on the `features` field, which is what we are interested in. It is an array of tables (here there is only one), each containing a possible combination of tag, scripts and languages activating the lookup. We can see that some ligatures (since we're looking at such a feature, as the `type` indicates) will be activated if a) `liga` is on and b) no script and language is specified (i.e. both are defaults) or the script is Cyrillic and the language is unspecified, and so on and so forth. (In the rest of this paper, I will often write that "this tag activates that lookup"; the reader should mentally add: "if script and language do not say otherwise".) There might be several such subtables in `features`, one per tag, since a feature might be implemented

[20] The second entry `pp_l_2_k_1` must be ignored; in fact, it shouldn't be there at all and the `lookup` field should be a string, not a table.

[21] What, the reader might exclaim, a glyph variant has a place in Unicode? Not really, no: 57,568, i.e. 0xE0E0, is in the "private use area", where font designers are free to put whatever they wish, and which generally contains alternate forms. Nevertheless, Unicode does contain some variants, such as the f-ligatures.

[22] The reader may have noticed that I use the word *feature* somewhat freely, indeed eschewing any technical definition, denoting vaguely but conveniently a typographic manipulation on the font.

[23] The table entry is at index 19 even though the lookup has 18 in its name (and lookups are indeed numbered consecutively), because lookup numbering starts at 0, whereas Lua tables prefer 1 as the first index (table length and the `ipairs` function are sensitive to that). So, whenever I mention lookup `ab_c_`x, the reader should look at index $x+1$ in the `gsub` or `gpos` tables.

by several tags: for instance, in Minion, "oldstyle" numbers (e.g. 123 instead of 123) are activated by the `onum` tag, but also by `smcp` (small caps) and `c2sc` (capitals to small caps, i.e. turning *uppercase* letters to small caps, whereas `smcp` affects lowercase letters), because those kinds of numbers are deemed mandatory with small caps by the designer. In such a case of multiple subtables, only one tag/script/language combination suffices for the lookup to be activated. It might also happen that no `features` table is present (for instance, the first two lookups in Libertine); that is the case for lookups that aren't tied to a tag but instead are called by other (contextual) lookups. We shall come back to that later.

One important yet easily missed property of language tags is that they are four-character strings (which is why I added unusual quotes around a few strings in the table), the last character being a space (except for `dflt`). On the other hand, you're not going to demand that users specify the language they want to use with a space at the end, so you have to add it yourself.

That tags interact with scripts and languages to activate (or not) lookups is an important feature in OpenType, allowing for a flexible interface. For instance, the `locl` tag can be activated in all circumstances, to activate variants only for the specified scripts and languages: given n sets of localized forms, there is thus no need for n tags. Another well-known example is the *fi* ligature, which shouldn't be used in Turkish, since this language distinguishes between *i* and ı, and the *i*'s dot is generally gobbled in the ligature.[24,25] Consequently, `liga` often points to two lookups: one with the *(f)fi* ligature(s) only, in which Turkish is excluded, and another implementing all other standard ligatures. That is exactly how the lookup shown here works: the reader may have noticed that for the Latin script, `TRK` is missing; and indeed that lookup controls the *(f)fi* ligature(s).[26]

One might think that tags are only pointers to lookups, which implement a feature, so that tags can be somewhat overlooked as simply being a user-friendly way to activate lookups, furthered by the fact that tags can be freely made up by font designers whereas there is only a finite number of lookup types. Although that is true in general, in some cases tags have some additional content. For one thing, some tags should be active by default (e.g. `liga`, standard ligatures) whereas others are optional (e.g. `hlig`, historical ligatures); this means that the former should be applied unless the user specifies otherwise.[27] Another, more important way in which tags have semantics is when the lookup they point to doesn't suffice to implement a feature properly. For instance, the `init`, `medi` and `fina` tags denote the initial, medial and final letterforms in (mostly) Arabic scripts; the feature is implemented by a simple substitution, not a contextual substitution (on which, see below). This means that if the user selects `init`, a given variant should be used for letters at the beginning of a word only, even though the lookup implementing it doesn't mention contexts; in other words, manipulations shouldn't rely on the lookup type only and should keep in mind the tag that activated it. A list of registered tags, along with their meaning, is available at [5].

Finally, we must note that the user's choice of script and/or language should be obeyed if and only if the font knows about them. Indeed, fonts do not specify all possible scripts and languages, but only those which make a difference; for instance, French isn't mentioned anywhere in Libertine, but this doesn't mean that no features should be applied if the font is loaded with `FRA` as the value of `lang` (and, of course, `latn` as the value of `script`). Instead, it means that the font knows nothing of that language and will treat it like the `dflt` language in the `latn` script. How do we know which scripts and languages a font is designed to handle? It is given nowhere in `fonttable`; instead, we must scan all lookups in `gsub` and `gpos` and collect all scripts and languages (actually, `gsub` and `gpos` should be treated independently of each other, but that often makes little difference). Only then can the user's request be interpreted correctly.

[24] Dotless *i* notes the close back unrounded vowel, which means it's at the back of the throat (like the Castle of Aaaaarrrgh), the mouth is only slightly open (unlike said Castle), and the lips are at rest; you can pronounce it more or less as follows: hold "oooooo" as in *boot* and stop pouting.

[25] The *fi* ligature does not always gobble the *i*'s dot; for instance it doesn't in Kris Holmes and Charles Bigelow's Lucida; accordingly, the ligature is applied even in Turkish.

[26] Another way to avoid the ligature in Turkish is illustrated in Minion: it has only one set of ligatures (containing *fi*) defined for all languages; however, it also has a special substitution activated by `locl` only if the language is Turkish (and similar), which replaces *i* with an identical glyph whose only difference is precisely not forming *fi* when preceded by *f*.

[27] What counts as a standard ligature (or any other feature) is up to the font designer; for instance, some fonts do not have the *fi* or *ff* ligatures (for the latter, see e.g. Adrian Frutiger's Frutiger), because they don't need it, whereas some less common ligatures are sometimes required: Libertine, for instance, has *Th*, *ch* and others.

9 Applying lookups

When several lookups are active, the order in which they are applied is crucial; for instance, if both `smcp` (small caps) and `liga` (ligatures) are active, the former should generally be executed before the latter, otherwise the sequence $f + i$ will be replaced with the ligature *fi*, which has no counterpart in small caps: and that's DEfiNITELY ugly.[28]

Fortunately, the order in which lookups should be applied is easily read from the font file and doesn't depend on any additional knowledge, as the previous example might imply. It works as follows (here "lookup" means "active lookup"): substitution lookups, i.e. those in the `gsub` table, are applied before the positioning lookups in `gpos`; in each category, lookups are applied in the order in which they are declared, i.e. according to their indices in `fonttable.gsub/gpos`. A lookup is applied to all glyphs in a node list before moving on to the next lookup (although "all glyphs" will be explicated below). Applying a lookup to one or more glyph(s) means that each subtable in the lookup is tried until one matches or no subtable remains (in which case the lookup simply does not apply).

These subtables are found (not so surprisingly) in the `subtables` entry. They generally contain only a single `name`, rather than what the subtable actually does: the latter is to be found in the glyph tables, where that information is associated with `name` (recall what was said above about the substitution of `f.short` for f and the kerning with o; we said that the substitution should occur if `ss_l_0_s` is active: `ss_l_0_s` is the `name` of a subtable in the `ss_l_0` lookup).

To sum up: the user specifies a script (or we use `DFLT`), a language (or `dflt`), and some tags (to which we add those activated by default). This combination activates lookups in `gsub` and `gpos`; with the names of those lookups' subtables, we retrieve what should be done by browsing the rest of the font (mostly, the glyphs); then, well, we do it. What should be done and how to do it is explained in the two sections below about lookup types.

For instance, let's suppose the user has selected `liga`, `kern` (kerning) and `mark` (mark positioning) in Libertine, and no script or language is specified. Further suppose that the input node list is:

fie`re che`vre

a French phrase (*fière chèvre*) meaning "proud goat" and interesting only to the extent that it lets us test our lookup-application skills. Here, ` is the character whose codepoint is 768, i.e. the "combining grave accent", not the "grave accent" at position 96. So, we begin with the `gsub` lookups, and `liga` activates two of them: `ls_l_18` (the one shown above) and `ls_l_19`; `ls_l_18` is applied to the beginning of the string, i.e. we test whether its only subtable matches, and since that subtable is supposed to turn $f + i$ into *fi*, indeed it does match: the first two glyphs are replaced by a single one representing the ligature; `ls_l_18` then moves to the third glyph,[29] obviously it doesn't match, so it moves to the fourth, which doesn't match either, and so on until the list has been scanned to the end. Enter `ls_l_19`: it is in charge of all remaining ligatures, and it matches on `ch` (see footnote 27) and only there.

Next come the `gpos` features; `kern` activates lookup 2, `pp_l_2`, and `mark` lookup 3, `mb_l_3`. The former is applied first, and has two subtables: thus, on each glyph, if the first matches, the second isn't applied, even though it might well have matched too. The first subtable deals with individual glyph pairs; the second with more general but less precise kerning tables (more details on this below). One could say that the second subtable defines default kerning, while the first deals with special cases. The order of their application thus makes sense.

So, for each glyph, `pp_l_2` will add some kerning if necessary. An interesting situation arises with our node list: there should be some kerning between e and v; however, the intervening accent, which the next lookup will properly place on the preceding letter, hides the fact that the two glyphs are next to each other. Or does it? No, it doesn't, for the precise reason that `pp_l_2` was instructed to ignore marks, and as we'll see later this accent belongs to that category; thus, when looking for an $e + v$ sequence, the lookup will operate as if the mark were not there. Finally, `mb_l_3` is applied, which has no less than seven subtables, one for each of the possible places where diacritics might be positioned relative to a base mark. In our case the fourth subtable will do the job, since it defines an anchor common to both e and the accent.

Applying lookups doesn't seem too complicated,

[28] Hermann Zapf's Palatino does have an *fi* (and others) to FI lookup. But since `smcp` is applied before `liga` anyway, it is useful only when the ligature has been entered by hand, e.g. as `\char"FB01`. The substitution is also in Test Libertine.

[29] Since *fi* has been processed, it can't be used as input any more for the current lookup. If the font defined a ligature between *fi* and *e*, then it should be implemented in a subsequent lookup, but the font is more likely to define a three-component ligature ($f + i + e$), which `ls_l_18` would have tried before *fi* because it is longer. More details in the subsection on ligatures below.

and indeed in many cases it isn't. Sometimes things are more difficult, though. For instance, given the node list "c¸h", we want it to form *çh*, i.e. we want the ligature with the cedilla on the first letter.[30] The ligaturing lookup is clever and ignores marks, so the ligature is properly formed; the mark positioning lookup is clever too and knows how to set a mark relative to one component of a ligature (e.g. a mark on *c* won't be placed like a mark on *h*). But the problem is that once the ligature is formed, and the mark has been put after it so that the positioning lookup can find it, how do we know where in the ligature the cedilla is to be positioned? Before ligaturing, it was clear: the cedilla was to be positioned on the first glyph; afterwards, that information was lost. Thus, you are in charge of passing this information from the ligaturing lookup to the positioning lookup; attributes can be used for this.

I've mentioned lookups ignoring marks. Indeed, glyphs in an OpenType font belong to one of four classes: base, mark, ligature, and component (of a ligature); glyphs also may have no class. Lookups can be instructed to ignore some of these classes (except the "component" class), in which case, when looking for possible input, they act as if glyphs of those classes were not present, so that *ABC* is seen as *AC* if *B* belongs to an ignored class. The class of a glyph is stored in the `class` entry of the glyph's table in `fonttable`, as we saw when inspecting *f* above.

A lookup's ignoring of some glyph classes is indicated in its `flags` subtable; for instance, `ls_l_19` in `gsub` has:

```
flags = { ignorecombiningmarks = true }
```

Thus that lookup, responsible for forming some ligatures, is blind to mark glyphs. Indeed, as we've seen, in "c¸h" the ligature will be created as expected. Besides `ignorecombiningmarks`, there are `ignorebaseglyphs` and `ignoreligatures` to ignore other classes. A lookup may also have another entry, `mark_class`, a string holding a mark class's name; in that case, the lookup ignores all marks except those in that class. To know which mark belongs to which class, we can inspect `fonttable.mark_classes`:

```
MarkClass-1 = gravecomb acutecomb uni0302
MarkClass-2 = cedilla dotbelowcomb uni0325
```

Thus, if `flags` above was:

```
flags = { mark_class = MarkClass-1 }
```

then the lookup would ignore all marks except the grave, acute and circumflex accents.

A lookup can have one last flag, `r2l` ("right to left"), which we will discuss briefly along with cursive positioning, since it is significant only with that lookup type.

10 Implementing lookups

There are several kinds of lookups, each specifying a particular implementation (modulo the remark above about tags that must also be taken into account). Most of them cannot be handled by LuaTEX by itself, meaning that we can't simply return a `metrics` table containing the information for performing the necessary operations. What LuaTEX implements natively is: non-contextual kerning, non-contextual two-glyph ligatures (although we'll later see a trick to create ligatures involving more than two glyphs), and non-contextual substitution (e.g. *f* in small caps; actually LuaTEX doesn't substitute anything, we simply return `metrics` with small-cap *f* at index 102). Other features must be done by hand in one of the pre-paragraph-building callbacks, i.e. `hyphenate`, `ligaturing`, `kerning` or `pre_linebreak_filter`[31] (see [3] for an introduction to what those callbacks do by default).

But even what LuaTEX can do is better done by hand; suppose that you have a contextual substitution (which LuaTEX can't handle) and a non-contextual substitution (which you can ask LuaTEX to perform by itself simply by returning `metrics` with the replacement glyphs at the position of the glyphs they replace); and further suppose that the former should happen before the latter (because it has a lower index in `fonttable.gsub`). You can never get that right, because LuaTEX will map characters to glyph nodes well before any of the above-mentioned callbacks are executed; in effect, the second substitution will have already occurred when the first is performed, thus reversing the desired order. Nonetheless, with e.g. the Latin script, non-contextual kerning and ligatures, and simple, across-the-board substitutions (like small caps), LuaTEX can be trusted, and you don't have to do anything by hand — in particular, you don't have to worry about hyphenation, a thorny issue as we'll see below; that's why I'll describe how some lookups can

[30] A *ç* can occur before an *h* in Manx, spoken (infrequently nowadays) on the Isle of Man. Nicolas Beauzée, a French grammarian of the XVIIIth century (often with modern insights) and contributor to Diderot and d'Alembert's *Encyclopédie*, suggested *çh* also be used in French orthography.

[31] The `linebreak_filter` (the paragraph builder proper) and `post_linebreak_filter` callbacks can be considered too for features related to justification, but I will not pursue that complex subject here. In addition, if anything is done in `pre_linebreak_filter`, it should be done in `hpack_filter` too, since the latter, but not the former, is called when material is processed in a horizontal box.

be implemented without extra work, even though I'll also give the bigger, callback-based picture.

In what follows, when explaining implementations with those callbacks, I will rely on the following assumption: that it is possible, given a font `id` (remember the third argument to `define_font`), to determine what should be done for each character in that font. It means that when a font is loaded, features to be implemented via callbacks should be stored somewhere. Also, applying lookups makes sense only for a sequence of glyphs sharing the same font; thus a paragraph made of

```
\myfont This is {\myotherfont very} special
```

would be processed in three independent steps, as if there were three paragraphs. A simple solution is to break the node list into sublists each with glyphs from the same font.

OpenType in general assumes that a text-processing application takes an input string, maps the characters to glyph positions in the font, and then manipulates those glyphs according to active lookups. the first part (characters to glyphs) is done automatically by LuaTEX, provided that at index m in `metrics.characters` there is the glyph at position n in `fonttable.glyphs`, with n being the value of entry m in `fonttable.map.map`. We've already done that, but there is an exception: TEX doesn't map a space character to a glyph, but creates a glue node instead.[32] Also, the list we'll be working on is interspersed with nodes which have absolutely no relation to characters, like penalties, whatsits, etc., so that we don't deal with a simple string of characters as assumed by the OpenType model. Finally, some of those non-glyph nodes are discretionaries, which require special treatment.

We solve the first problem by considering a glue node to be a character whose glyph is `fonttable.map.map[32]` (since 32 is the space's codepoint); this might seem like overkill, but it is not: lookups do take spaces into account, and Libertine has kerning pairs with space as either the first or the second member. Compare:

How Terrible!
How Terrible!

In the first line the space has its default width, whereas in the second line (negative) kern is added after w and before T (while that kerning is not applied between other letters). In Minion, space is subject in some circumstances to substitution (being replaced with a narrower space glyph).[33] However, some lookups (or rather, tags, such as `init` above) require that a "word" be a well-defined entity, because position in the word is important; in that case, spaces should be considered boundaries.

The second problem, non-glyph nodes, is solved just by ignoring those nodes. Thus, in this node list:
⟨*glyph node 1*⟩ ⟨*non-glyph node*⟩ ⟨*glyph node 2*⟩
a function asked to retrieve the glyph following the first one should return the third node. Things might get a little bit more complicated, but one shouldn't worry too much; special circumstances may simply demand special solutions.

The third problem is harder, because we definitely can't ignore discretionary nodes. Suppose you have the string `X{a}{b}{c}Y`, where `X` and `Y` can be anything, `a` is the `pre` field of a discretionary (i.e. the part typeset before the break if hyphenation occurs at this point; typically, a hyphen), `b` is the `post` field (the part typeset at the beginning of the following line if hyphenation occurs; typically empty) and `c` the `replace` field (what is typeset if the hyphenation point is not chosen; also typically empty). (See [3] for a slightly more detailed description of discretionary nodes.) You may have lookups acting on the strings `Xa`, `bY` and `XcY`, so that each lookup should be applied three times, and the resulting material should be `{a'}{b'}{c'}`, where `a'`, `b'` and `c'` are respectively `Xa`, `bY` and `XcY` after lookups have been applied. Then any common part at the beginning of `a'` and `c'` can be moved out of and before the discretionary; likewise, common material at the end of `b'` and `c'` can be moved after. Consider the following extremely hypothetical, not to

[32] Of course TEX does this only because space has catcode 10; give it catcode 12, and it will be treated as a glyph. But you don't want to do that, do you?

[33] What shall we do in this case? We'll see below that we implement substitution between glyph nodes; shall we replace the glue node here with a glyph node? If the replacement is a real glyph, then yes, we can't do otherwise; but if it is another space glyph, we will adjust the original glue node's width (stretch and shrink are of course irrelevant to OpenType and should be dealt with as best seen fit). A space glyph can be spotted quite easily: all its `boundingbox` values are 0. This does not prevent it from having a positive `width` field, though, since a glyph's width does not depend on its shape.

Another problem is that not all glue nodes come from space characters (as I write this, LuaTEX does not distinguish between glue from a space character and glue from an `\hskip`), and kerns should also be considered as space characters. But lookups on space glyphs crucially depend on the space's shape, i.e. its width; such lookups are generally kerning pairs (even Minion's case is kerning in disguise) meant to yield a homogenous space between all pairs of letters — especially important in flowery script fonts — but what if two letters are already separated by an important glue or kern? Should additional kerning be performed? In other words, how to solve the mismatch between what the user wants and what the font requires? I have no answer to that.

mention extremely dumb, example:

```
V{a-}{T}{o}e
```

This gives `Voe` if no hyphenation occurs, and `Va-Te` otherwise. Provided it is typeset with Computer Modern, there should be some kerning between `V` and `a` or `o`, `T` and `e`, and `o` and `e`, i.e. using ⟨*k*⟩ to denote a kern node, without hyphenation we get `V`⟨*k*⟩`o`⟨*k*⟩`e`, and with hyphenation `V`⟨*k*⟩`a-T`⟨*k*⟩`e`. So our string first becomes:

```
{Va-}{Te}{Voe}
```

Then, after applying the kerning lookup:

`{V`⟨*k*⟩`a-}{T`⟨*k*⟩`e}{V`⟨*k*⟩`o`⟨*k*⟩`e}`

And now, moving out common parts:

`V{`⟨*k*⟩`a-}{T`⟨*k*⟩`}{`⟨*k*⟩`o`⟨*k*⟩`}e`

The kerns have *not* been moved, because they're different in each part.[34]

Now, this is all easier said than done (and it wasn't that easy to say). And it still doesn't solve some problems, for one, what `X` and `Y` actually are depends on the lookup under investigation; e.g. a typical ligature would set `X` as `f` and `Y` as `fi` in `of{-}{}{}fice`, resulting in `o{f-}{`⟨*fi*⟩`}{`⟨*ffi*⟩`}ce`. Worse, this does not even begin to consider what happens if a lookup spans several discretionaries (as in `of{-}{}{}f{-}{}{}ice`), in which case LuaTEX becomes really perverted. We'll leave the issue at that — and as an exercise to the most courageous readers only; suffice it to say here that you might have to "flatten" a few discretionaries, i.e. simply remove them and put their `replace` fields in their place.

11 Lookup types: substitutions

There are two main families of lookups: those related to substitutions, and those related to positioning. The conditions for the former, as illustrated in the preceding section (i.e. the tag/script/language combination), are found in the `fonttable.gsub` table, and those for the latter in `fonttable.gpos`. What a lookup *does* is generally found in the affected glyph(s), with some exceptions. I will review both lookup types: first substitutions, then positioning.

Single substitution. This kind of lookup replaces one glyph with another glyph: e.g. small caps, "old-style" numerals, etc. As an example, Libertine replaces ʂ with ʂ[35] if the `locl` feature is on and the language is Romanian or Moldavian (and the script, of course, is Latin):

```
name = ss_latn_l_7
type = gsub_single
flags = { }
subtables = { 1 = { name = ss_latn_l_7_s } }
features = {
  1 = { tag = locl
        scripts = { 1 = {
                        script = latn
                        langs  = { 1 = MOL
                                   2 = ROM } } } } }
```

And indeed if we look at some selected portion of the `lookups` table in ʂ (350), we'll see:

```
ss_latn_l_7_s = {
  1 = { type = substitution
        specification = {
          variant = Scommaaccent } } }
```

In both tables, the `type` tells us we're dealing with a single substitution; the `specification` subtable will occur in many different types of lookups, but the entries it contains will be different. Here it's quite straightforward: the `variant` entry points to the replacement.

Implementing a single substitution is easy: you just change the `char` field of the node under investigation, and its dimensions (`width`, `height` and `depth`) follow suit. If we want to let LuaTEX do that by itself, though, we can do the following when loading the font:

```
metrics.characters[350] = metrics.characters[536]
name_to_unicode["Scommaaccent"] = 350
```

But this is only the tip of the iceberg. Suppose for instance that we're implementing small caps this way: at position 104, letter *h*, we'll put small-cap *h*. Now, both *T* and *c* form a ligature with *h* in Libertine; *c* will be replaced with its small-cap counterpart, so the ligature will vanish; but *T* (under the assumption that ligatures are handled by LuaTEX too) will be instructed to form a ligature with character 104 (since `metrics` is only interested in codepoints), the result being: *Ths*. To avoid that, each time something refers to *h* in the original `metrics`, it should be made to refer to small-cap *h* instead. That is not particularly difficult if `name_to_unicode` is properly kept up to date. But single substitutions aren't so innocent anyway; beside the ordering problem mentioned above, single substitutions can be difficult because, as already said, they may requiring analyzing the context, as is the case with `init` and associates, which replaces glyphs in some positions

[34] The user can check that the last code snippet is what LuaTEX returns when left to its own devices by scanning the node list in the `pre_linebreak_filter` callback for a paragraph made of `V\discretionary{a-}{T}{o}e`.

[35] Denoting a voiceless postalveolar fricative commonly used to *shush* people. Put your tongue against the back of your upper teeth; move it a bit toward your palate, you'll feel a ridge there; move it again: that's your postalveolar region (the ridge itself is the alveolar region). A fricative is a consonant whose sound is the air going through an opening so narrow that it generates turbulence.

only. Also, I've said that a tag may be associated with more than one lookup; what I've not said is that those lookups can very well belong to different types, and they do not even need to be all substitutions or positioning; for instance, `onum` in Libertine points to a single substitution (see `ss_l_25`) but also to a positioning lookup in `gpos` (`sp_l_1`) that lowers mathematical operators. So you can leave the substitution to LuaTEX, but you'll still apply the second lookup yourself.

Ligatures. A ligature is the replacement of two or more glyphs with a single one. In TEX, such a replacement is implemented as information in the first character in the ligature, more precisely in its `ligatures` table; for instance *f* (102) in Don Knuth's Computer Modern has the following:[36]

```
ligatures = {
  102 = { char = 11, type = 0 }
  105 = { char = 12, type = 0 }
  108 = { char = 13, type = 0 } }
```

We're looking at the *ff*, *fi* and *fl* ligatures respectively; if TEX does ligaturing by itself, whenever a node with `char` 102 (*f*) precedes a node with `char` 105 (*i*), it will replace them with a node whose `char` is 12, and at position 12 in `cmr10` is *fi*. The `type` we shall ignore.[37]

But where are the *ffi* and *ffl* ligatures? They are stored in the *ff* character (11). Indeed, for TEX, those two ligatures are formed by *ff* + *i*/*l*, not by *f* + *f* + *i*/*l*,[38] because a ligature always involves two characters. OpenType fonts, on the other hand, have no upper limit on the number of components of a ligature, and even though *ffi* might be (and sometimes is) described as a ligature between *ff* and *i*, it can also (and generally is) implemented as involving three glyphs. So what? Can't we just write a few lines of code so that *f* + *f* + *i* = *ffi* in `fonttable` is split into *f* + *f* = *ff* and *ff* + *i* = *ffi* in `metrics`? We can do that; more generally, we could turn a ligature L with n components into $n-1$ ligatures with two components as follows:

$l_1 = c_1 + c_2$
$l_2 = l_1 + c_3$
...
$l_{n-1} = l_{n-2} + c_n = L$

The problem is: what if l_i doesn't exist as an independent glyph? The *ffi* ligature is misleading, because there does exist an *ff* ligature. But that is not always the case, quite the contrary. As an example, the `frac` tag in Libertine defines ½ as *1* + */* + *2*, and there is obviously no *1/* glyph. If we proceed carefully, we could define "phantom" characters such as *1/*, whose only purpose in life is to form a ligature with the subsequent character; but then we'll run into trouble if there is no such character, as in:

```
An enumeration:
1/ First ...;
2/ then ...;
```

So the method might work for standard ligatures in a Latin script, where the ligatures with three components are the ones described above, but apart from that it's better to use callbacks.

But let's get back to `fonttable` for a while; I've already shown a ligature lookup above (`ls_l_18`). The only thing we haven't remarked upon yet is the `type`, which is, not surprisingly, `gsub_ligature`. Ligatures themselves are to be found in the replacement glyphs, not in the first glyph as in TEX; thus *f* + *f* + *i* is stored in *ffi* (64, 259):

```
name    = f_f_i
unicode = 64259
class   = ligature
width   = 819
boundingbox = { ... }
lookups = {
  1 = { type = lcaret
        specification = { 1 = 266, 2 = 538 } } }
  ls_l_18 = {
    1 = { type = ligature
          specification = {
            char = f_f_i,
            components = f f i } } }
```

The `class` says `ligature` but it may very well be absent; glyph classes aren't mandatory. Hence, what informs us unambiguously that we're dealing with a ligature is the `ls_l_18_s` lookup and its `type`. As with single substitutions, the relevant information is stored in `specification`, and what we're really looking for is `components`, a string of glyph

[36] We are seeing the equivalent of `metrics` for `cmr10` here. Recall that our job is to turn `fonttable` into `metrics` and return it to LuaTEX; but all fonts are implemented with such a table, even if they were loaded automatically, i.e. nothing was registered in `define_font` and a TFM font was used. The table is returned by the `font.getfont` function, which takes as its single argument a font id. Font id's themselves are given by `font.id`, which is passed the control sequence (without the escape character) associated with the font. So what is shown here is `font.getfont(font.id("tenrm")).characters[102].ligatures` (with plain TEX, where `cmr10` was loaded with `\font\tenrm=cmr10`).

[37] But let's explain briefly: it specifies what the output of the ligature should be. 0 means that only the ligature glyph is returned, but one of the original nodes, or both, could also be retained (e.g. *f* + *i* would produce *fi*); `type` also determines where TEX should continue its ligaturing.

[38] I said in note 29 that once the initial *fi* has been formed, it can't be reused to form another ligature with the following character. I was then talking about the rules for OpenType lookups. TEX, on the other hand, does restart ligaturing at the newly formed ligature itself (unless `type` says otherwise), and *ff* is available to form a ligature with what follows.

names separated by space (glyph names themselves can never contain space).

While we're at it, the first subtable in `lookups` is not really a lookup; its index isn't a lookup name (a string) but a simple number, and it specifies where the caret should be placed to highlight individual components in the ligature. Since PDF has no support for this anyway, we shall ignore it.

Implementing ligatures in LuaTEX can be done as follows: we store all (active) ligatures into categories headed by the first component. Then, when scanning a node list, we check whether any ligature begins with the character of the node under investigation, and if so, whether the following nodes also match the other components. In doing so, it is crucial that ligatures be ordered by length (i.e. number of components) and that longer ligatures be tried before shorter one; otherwise, $f + f + i$ might be turned into $ff + i$ and stop there.

Identifying a string of nodes might require skipping over other nodes, if only because the lookups are instructed to ignore some glyph classes; the ligaturing lookup `ls_l_19_s`, for instance, should ignore marks, as discussed above. Once component nodes are properly identified, there are two ways to proceed. The easy way is: delete all component nodes except the first, whose `char` field you set to the ligature's codepoint, as in a single substitution. The best way is (though it makes a difference only when TEX reports information): remove all component nodes, replacing the first with a new glyph node with subtype 2 (ligature); arrange the removed nodes in a proper list (i.e. node i has `prev` set to node $i-1$ and `next` to node $i+1$, except the first node's `prev` field and the last node's `next` are both `nil`), and set the new node's `components` field to the first removed node. For instance, given three component nodes `n1`, `n2` and `n3` in list `head`:

```
local lig = node.new("glyph", 2)
lig.font, lig.attr = n1.font, n1.attr
lig.lang, lig.uchyph = n1.lang, n1.uchyph
lig.char = ⟨ligature codepoint⟩
head = node.insert_before(head, n1, lig)
node.remove(head, n1)
node.remove(head, n2)
node.remove(head, n3)
n1.prev, n1.next = nil, n2
n2.prev, n2.next = n1, n3
n3.prev, n3.next = n2, nil
lig.components   = n1
```

We set the ligature node's `font` (of course), `attr` (for attributes), `lang` and `uchyph` (both for hyphenation) fields to the original node's values, because they store important information.

If nodes were skipped, they will now occur after the ligature node. If those nodes were mark glyphs, then they should be marked (no pun intended) with an attribute so that they can later be placed on the correct component of the ligature (a subject we address more thoroughly below).

Multiple substitution. This type of lookup is the inverse of the previous one: one glyph is replaced with several glyphs.[39] In the `gsub` table, this lookup has `type` set to `gsub_multiple`; in the affected glyph, `type` is `multiple`, and `specification` has a single `components` entry similar to that of a ligature, i.e. a list of glyphs separated by spaces.

What should be done on the TEX side is just the reverse of the previous lookup type; since there is nothing particularly instructive in that, I leave it as an exercise to the reader. The f-ligatures in Libertine all have multiple substitution when `smcp` is on (see `ms_l_12`).

Alternate substitution. This maps a single glyph to one or more variants (often, but not always, used in other features); for instance, R (82) in Libertine is mapped to a stylistic alternative (otherwise tied to the `ss02` tag) and a small capital (from `c2sc`). In `gsub`, the lookup has type `gsub_alternate`; in the glyph, `type` is `alternate` and `specification` contains a single `components` entry, a string with one or more glyph names separated by space.

This kind of lookup is typically activated with the `aalt` (*Access All Alternates*) tag, through which the user may choose a variant for a glyph, even though the conditions required for this variant to occur aren't met (e.g. you want the medial form of a glyph even though you're not in the middle of a word). A graphical interface with a drop-down list is obviously better fitted than TEX to do that, although one can easily design a `\usevariant{`⟨*number*⟩`}{`⟨*glyph*⟩`}` command.

But alternate substitutions can be put to better effect with another tag, `rand`, for selecting a glyph variant at random. Given the `components` list, one can decide which variant to use via Lua's `math.random` function. Once this is done, the replacement is identical to a single substitution. This feature can be found in the Punk Nova font by Hans Hagen and Taco Hoekwater (after Don Knuth).

[39] I've been telling a white lie: "ligatures" can take a single glyph as input, and "multiple substitutions" can output a single glyph too. In both case, they're equivalent to single substitutions. So those lookup types are better described as respectively "n glyph(s) to 1 glyph" and "1 glyph to n glyph(s)" replacements, with $n \geq 1$. That doesn't change the import of what is described in the main text. See the slash (47) in Libertine for an example of a "one-glyph ligature".

Contextual substitution. Those last couple of lookup types weren't very exciting. But now, fasten your seat-belt, we're in for the real thing: contextual substitution, and chaining contextual substitution, and finally reverse chaining contextual single substitution, all that possibly expressed in three different formats ... Alas, the terribly-named reverse chaining contextual single substitution isn't supported in LuaTEX (or virtually anywhere else), so we won't be studying it.[40]

Up to now, we've seen lookups identifying input and replacing it with some output. Contextual lookups also identify input, but then they call other lookups on parts of that input to do the substitutions. For instance, given input `ABCD`, `A` may be turned to a small cap, `BC` may form a ligature, whereas nothing happens to `D` but its being there is still crucial to identify the proper input, i.e. otherwise the former two substitutions wouldn't have been performed.

As alluded to above, contextual lookups may be expressed in three formats. For contextual substitutions proper, we'll see the glyph-based format; in the subsection below about chaining contextual substitution, class-based and coverage-based formats will be investigated; but all three formats can be used with both types of lookup.

Libertine has a tag called `gtex` (for *Greek TEX*) which turns the input sequence `TeX` into $\tau\varepsilon\chi$ (with lowercase *tau* and *chi* so they won't be confused with T and X). Of course, you don't want to change every T, e and X to tau, epsilon and chi respectively; rather, you want the substitution to be performed only when the three letters are ordered as in `TeX`; in other words, you want a contextual substitution.

If we look into `cs_l_4` — the lookups associated with `gtex` — in `fonttable.gsub`, we won't learn anything we don't already know, except that it has type `gsub_context`. While we're at it, we can retrieve the name of its only subtable: `cs_l_4_s`. We won't find what this subtable does among the glyphs, as with the previous lookup types; instead, contextual lookups live in their own table, `fonttable.lookups`, where they are indexed by name. So, let's look at `fonttable.lookups.cs_l_4_s`:

```
type   = contextsub
format = glyphs
rules  = {
  1 = { glyphs  = { names = T e X }
        lookups = { 1 = ss_l_2
                    2 = ss_l_2
                    3 = ss_l_2 } } }
```

The important information is contained in the `rules` subtable; it describes the context and what to do with it. But in order to understand that subtable, the value of `format` is crucial: it tells us how the context will be defined. The `rules` table is made of subtables at consecutive indices, each defining a context and what to do with it, somewhat like subtables in a lookup of the types seen before: the first subtable that matches wins the prize.

The context itself is defined in `glyphs`, which contains at least a `names` subtable, and if it is a chaining contextual substitution, perhaps also `back` and `fore` fields. Those three fields are strings made of glyph names that denote an input sequence (space is used to delimit names only), somewhat like the `components` entry of a ligature substitution. Here, then (it goes without saying but let's say it anyway), the relevant input string is `TeX`.

The `lookups` table lists what should be done to the successful input sequence. It is not a simple array; instead, the indices correspond to positions in the input sequence. Here the table reads: apply lookup `ss_l_2` to the glyph at position 1; then apply it again at position 2 and 3. If we explore `ss_l_2` in `fonttable.gsub`, we'll see that it's a single substitution, and in the tables for T (84), e (101) and X (88), we'll find that it maps those glyphs to their Greek counterparts. In other words, `ss_l_2` is executed like any other lookup, except that, each time it is called, it inspects only one position in the node list, not the entire list.[41]

A crucial point is that each index in `lookups` identifies a position in the input sequence *after* the previous lookup has been applied. In our example, this doesn't make any difference, but suppose a contextual lookup is designed to match against `ABC`, and suppose the first lookup it calls (at index 1) is a ligature, so that `AB` is turned into `X`; then the sequence is now `XC`, and if a second lookup is called to act on `C`, it will have index 2, not 3, even though `C` originally was at position 3.

Also, the `lookups` table isn't mandatory: a contextual lookup may very well identify a valid input sequence and do nothing with it. Below we'll see a

[40] The main difference between reverse [...] substitution and the others is that it processes the node list starting at the end; e.g. given `ABC`, first it deals with `C`, then `B`, then `A`. This has nothing to do with the direction of writing, though it was meant for Arabic calligraphy, where a glyph variant may be determined by the shape of the following glyph (as in the Nasta'līq style), so that the latter should be set before the former. Nonetheless, none of the Arabic fonts I've seen use reverse [...] substitution.

[41] More accurately, the lookup inspects input from one position only, even though it might need to inspect several glyphs, as in a ligature.

detailed example of that, but here we can say that it can be used to prevent a subsequent subtable from matching: for instance, if you want to define context ABX as AB followed by any glyph but, say, z, then you can define a contextual lookup with a first subtable matching ABz and doing nothing, and the second simply as AB, i.e. matching independently of what follows.

How shall we implement a contextual substitution? Identifying the input can be done as for a ligature; then each lookup is applied as usual, albeit on a particular node (according to the lookup's index), which can be ensured simply by counting glyph nodes. The only subtlety is that lookups should be applied in order, i.e. lookup at index m should be executed before lookup at index n, with $m < n$. In our example, since indices in `lookups` are consecutive, a simple `ipairs` will respect the ordering; however, there might not be a lookup at each index, and `lookups` could have entries at indices `1`, `2` and `4`, for instance, in which case `ipairs` will stop at `2` whereas `pairs` won't iterate in any particular order. Thus one should create another table reflecting the original order but also retaining the positions:

```
local t = {}
for i, l in pairs(lookups) do
  table.insert(t, {position = i, lookup = l})
end
table.sort(t, function (a,b)
                return a.position < b.position end)
```

Now we can traverse `t` with `ipairs` and apply the lookups in order.

Chaining contextual substitution. Contextual substitutions, like other substitutions, completely consume their inputs, even if lookups are applied at some positions only. For instance, a contextual substitution acting on `TeX` and simply turning `e` to epsilon will nonetheless consider `X` as processed; the next iteration of the current lookup will begin at the next character. Also, since what came before the current position is considered processed too, contextual substitutions can't take that into account either. In other words, substitutions of that type (and of all the types seen up to now) have no memory, and they can't foresee the future.

A chaining contextual substitution, on the other hand, is precisely that: a contextual substitution with memory and foresight. In other words, it can take into account already-processed glyphs and future glyphs without consuming the latter. Chaining contextual substitutions are so useful that virtually all fonts use them even when simple contextual substitutions would do.

In essence, a chaining contextual substitution works like a simple contextual substitution: it identifies a sequence of glyphs and calls other lookups at some positions in this sequence. But it can also identify preceding glyphs, called the *backtrack sequence*, and/or subsequent glyphs, called the *lookahead sequence*.

An important point to keep in mind is that, whatever the format, the backtrack sequence is always set in reverse order of the direction of writing; for instance, to identify `abc[xyz]def`, where `abc` and `def` are the backtrack and lookahead sequences respectively, and `xyz` the input sequence proper, a lookup in the `glyphs` format (as in the previous section) would have the `glyphs` table organized as:

```
back  = c b a
names = x y z
fore  = d e f
```

Note how the `back`track sequence is displayed.

In Libertine, the `ccmp` tag (*Glyph Composition/Decomposition*, a somewhat all-purpose tag) activates a lookup, `ks_l_32`, with two subtables. After noting the lookup's `type` (`gsub_contextchain`), let's look at `ks_l_32_c_0`, the first of those two subtables, in `fonttable.lookups`:

```
type = chainsub
format = coverage
rules = {
  1 = {
    lookups  = { 1 = ss_l_0 }
    coverage = {
      current = { 1 = f f_f }
      after   = {
        1 = parenright question T ... } } } }
```

This subtable uses the `coverage` format, which specifies a set of glyphs for each position in the input, backtrack, and lookahead sequences, denoted respectively by subtables called `current`, `before` (missing here) and `after`, whose indices are the position in the sequences (starting from the end in `before`) and the values at those indices are strings representing the glyphs. In this example, the input sequence is made of one glyph, either *f* or *ff*, and the lookahead sequence also contains one glyph, which can be a question mark, a right parenthesis, *T*, and many others elided here. So, *f?, f), fT, ff?* ... will all be identified by this single rule. This works a bit like a regular expression (with `f_f` denoting the ligature): `[ff_f][?)T...]`. If one of the three sequences had contained more than one glyph, it would have had an equal number of entries.

The `lookups` table should be read as in a simple contextual substitution, with indices denoting positions in `current`. Here, `ss_l_0` should be applied at position one; this lookup replaces *f* (both on its

own and in the ligature) with a variant whose arm is shorter, so it doesn't touch the next glyph: *f?* becomes *f?*.[42]

One last remark about our example: although the relevant context here is described at entry 1 in `rules`, there can be no other entries in that table, since the `coverage` format allows only one context per subtable. Hence another context would be defined in another subtable. That is not true of the other two formats, `glyphs` and `class`.

Let's turn to an example of the latter. Libertine can turn *etc.* to *&c.* thanks to the `etca` tag (ET CA*etera*, or ETC. *with Ampersand*, or ETC. *Alternate*, or ETC. with a dummy letter only because tags should be made of four characters; I did *not* spend hours trying to find a meaningful name). The core of the feature is a simple ligature ($e + t$ to &), but performed if and only if the two letters are at the beginning of a word (to exclude text like *fetch*), and followed by c and a period (so *etch* is also excluded).[43]

The `etca` tag points to lookup `ks_l_5`, whose only subtable `ks_l_5_s` is (in `fonttable.lookups`):

```
type          = chainsub
format        = class
before_class  = { 1 = space parenleft bracketleft }
current_class = { 1 = e, 2 = t }
after_class   = { 1 = c, 2 = period }
rules = {
  1 = { class = {
          before  = { 1 = 0 }
          current = { 1 = 1, 2 = 2 } } }
  2 = { class = {
          current = { 1 = 1, 2 = 2 }
            after = { 1 = 1, 2 = 2 } }
        lookups = { 1 = ls_l_3 } } }
```

As expected, `format` is `class`; but what is a class? First, it has absolutely nothing to do with the glyph classes mentioned above. Here a class is simply a set of glyphs created especially for a lookup. More precisely, classes are defined separately for the input, backtrack and lookahead sequences in (respectively) `current_class`, `before_class` and `after_class`. In each of those tables, a class is denoted by its index, and its content is a string of space-separated glyph names. For instance, class 1 for the backtrack sequence here contains space (recall that we said that space is a glyph in OpenType fonts), left parenthesis and left bracket, while the classes in the other sequences are singletons. In each subtable of `rules`, the context is described analogously to the `coverage` format: indices in the `current`, `before` and `after` subtables denote positions, but here the values point to classes (one class per position); for instance, at position 1 in the input sequence for the first rule, there should be a member of `class` 1 for that sequence, i.e. an *e*. This is, again, a bit like regular expressions, except that sets `[...]` are assigned to variables beforehand and the regexp is defined with those variables.

Classes are a bit special in that they do not intersect, i.e. a glyph belongs to one and only one class (for each sequence, that is). This fact is of little value to us (though quite important to the font designer), except when paired with another one: that there exists a default class, which need not be defined and whose index is 0; this class contains all glyphs, except those present in other classes. Thus, class 0 in `before_class` contains everything but space, left parenthesis and left bracket, class 0 in `current_class` contains everything but e and t, and class 0 in `after_class` contains everything but c and period.

Class 0 is put to good use in rule 1; this rule matches any sequence `Xet` where `X` is anything but `(`, `[` or a space, since class 0 contains all glyphs but them. For instance, if we're inspecting `fetch`, rule 1 will match, preventing rule 2 from being applied at the same position — and that's the only reason why rule 1 exists at all: to prevent rule 2 from being applied in most circumstances; accordingly, rule 1 has no `lookups` table. However, rule 1 will not match with `(et` or `[et` ␣`et` (no matter what follows), since the initial glyph in each case doesn't belong to class 0, and rule 2 will perhaps match if the right sequence follows.

The implementation of a chaining contextual substitution is similar to that of a simple contextual substitution, except that we may request surrounding glyphs to identify themselves. No matter how many nodes we scan to compare them to the lookahead sequence, they do not count as processed, and the next iteration of the lookup starts at the glyph to the immediate right of the input sequence; in our example, provided the right context has been found, the lookup will start again at *c*.[44]

At this point, the reader might have a question nagging at the back of his or her mind, namely:

[42] The second subtable in `ks_l_32` substitutes *ı* and *ȷ* for *i* and *j* before accents, which will then be properly positioned.

[43] No word in English ends with *etc*, so the second condition would be enough; but you never know.

[44] Of course here it is impossible for the lookup to match on *c*, and it could have been skipped (i.e. the following *c* and period could very well have been parts of the input — not lookahead — sequence). However, the ampersand substitution could have been part of a much more general lookup implementing a wide range of stylistic variants, one of which might take *c* as its input.

what is the difference between a contextual lookup (chaining or not) with n subtables containing one rule each, as illustrated by `ccmp` above, and the same lookup with one subtable containing n rules, as `etca` here? As far as what the lookup *does* is concerned, the answer is: none, since a subtable matches when one of its rules matches, and a successful subtable or rule prevents the other ones from being applied. But there are some technical differences which may dictate why one implementation is chosen over the other: first, as already mentioned, only one rule per subtable is allowed in the `coverage` format, so a lookup will necessarily use several subtables; second, all rules in a subtable are in the same format (as witnessed by the fact that the `format` entry is on the same level as the `rules` table, and thus holds for all the rules contained in the latter), whereas different subtables in the same lookup may be expressed in different formats; if two contexts are better expressed in two different formats, then different subtables can be used.

12 Lookup types: positioning

At this point, we're done with substitutions (since reverse substitution is in limbo for the present). We turn to positioning lookups, held in the `fonttable.gpos` table.

Single positioning. This kind of feature moves a glyph horizontally and/or vertically and modifies its horizontal and/or vertical advance. Typographically, that means modifying the glyph's sidebearings (moving a glyph left/right increases/decreases the left sidebearing, and increasing/decreasing its width does the same for the right sidebearing). For instance, capital spacing in Libertine (activated by the `cpsp` tag) is implemented by lookup `sp_l_0` (with type `gpos_single`), whose only subtable `sp_l_0_s` is detailed in each uppercase glyph's `lookups` table, e.g. *A* (65):

```
sp_l_0_s = { 1 = {
    type = position
    specification = { x = 2, h = 5 } } }
```

As usual, what should be done is detailed in the `specification` subtable. For single positioning, the table may have up to four values: `x` is the horizontal displacement of the glyph; `h` is its width correction; `y` and `v` are the same things in the vertical direction.

Now suppose we're implementing capital spacing. We browse all capitals in a node list, or rather (and more generally), all glyphs that have this particular lookup. For each such glyph we add a kern of `x` units before if it follows a similar glyph, and a kern of `h` units after if it precedes a similar glyph (the *if*-clauses translate the fact that space adjustment should take place only *between* capitals). We could also merge the two kerns for each pair.

A caveat: capital spacing does not replace other forms of kerning, particularly the default kerning (denoted by the `kern` tag). If that kerning is done automatically by LuaTEX (because the `kerning` callback is left untouched or the `node.kerning` function is used), that should occur before capital spacing, otherwise our additional kerns will prevent it. In turn, when looking for the preceding or following glyph, we should ignore intervening kerns (which have the special subtype 0).

An example of vertical positioning can be found in `sp_l_1` (triggered by `onum`, whose only subtable `sp_l_1_s` lowers mathematical operators so they are better placed with "oldstyle" numbers). Thus the `specification` for this subtable in glyphs =, +, −, × and ÷ (61, 43, 8722, 215 and 247 respectively) contains a single entry at `y` whose value is `-100`; then it suffices to assign (or rather, add) −100 units to the glyph nodes' `yoffset` field to implement the lookup. I leave it at that here, since `yoffset` is detailed more thoroughly below, along with its horizontal twin `xoffset`.

Pair positioning (kerning). This second type of adjustment is well known: it is the tiny amount of space (possibly negative) added between two letters that look badly set when just left next to each other, for instance between *T* and *o* (compare kerned *To* with "natural" *To*).

Kerning pairs go in two formats. First, there is kerning information for individual pairs, which is found in the glyph table for the left member of the pair, more precisely in the `kerns` (not `lookups`) subtable. For instance, looking precisely at this subtable for *T* (84):

```
kerns = {
 1 = { char = udieresis
       off  = -44
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 2 = { char = odieresis
       off  = -44
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 3 = { char = adieresis
       off  = -36
       lookup = { 1=pp_l_2_g_0, 2=pp_l_2_k_1 } }
 ... 22 more
 }
```

This means that *ü* and *ö* should be brought closer to a preceding *T* by 44 units, whereas for *ä* it should be 36 units.[45]

[45] The `lookup` table shouldn't be a table at all, but a string identical to the table's first entry, i.e. `pp_l_2_g_0`; this hap-

However, different pairs often share the same amount of kerning. Better yet, classes of glyphs on the left will often have the same kerning when followed by classes of glyphs on the right. For instance, *A*, *À*, *Á* ... should behave similarly, because here accents make little difference (this is not true for a lowercase letter). Accordingly, kern pairs are often defined in kern tables; for instance, here's the entry at index 3 in `fonttable.gpos` (much abridged):

```
3 = {
  type = gpos_pair
  flags = { ignorecombiningmarks = true }
  name = pp_l_2
  features = { ... }
  subtables = {
    1 = { name = pp_l_2_g_0 }
    2 = {
      name = pp_l_2_k_1
      kernclass = {
        1 = {
          firsts = {
            2 = r v w y yacute ydieresis ...
            3 = a i u ...
            ...  }
          seconds = {
            2 = a aogonek
            3 = c e o q ccedilla
            ...
            11 = exclam parenright ...  }
          offsets = { 13 = -15
                      14 = -10
                      18 = -29
                      ...  }
          lookup = pp_l_2_k_1  } } } } }
```

We discover *en passant* that such kerning is associated with the `kern` tag, which activates both the per-glyph kerning (see the first subtable with lookup `pp_l_2_g_0` — also associated with the individual kerning pairs for *T* in the previous code snippet) we've seen above and the kerning by class that we're interested in; it is of course crucial that the per-glyph subtable take precedence over the kerning table: the latter is powerful yet indiscriminate, while the former is limited but accurate. Kerning tables deal with glyphs massively, and individual kerning pairs set the exceptions.

The `kernclass` table has the information we need, namely a kerning table,[46] composed of four entries: `firsts` is an array of glyph classes on the left; `seconds` is an array of glyph classes on the right; `offset` is the amount of kern needed for each pair of classes; and `lookup` is just redundant; we can ignore it and stick to the `name` field. Let f_i, s_j, o_k stand for entries `firsts[i]`, `seconds[j]` and `offsets[k]` respectively; then kerning tables can be read as follows:

	s_1	s_2	...	s_n
f_1	o_1	o_2	...	o_n
f_2	o_{n+1}	o_{n+2}	...	o_{n+n}
...	...	...	...	...
f_m	o_{mn-n+1}	o_{mn-n+2}	...	o_{mn}

In words, the amount of kerning between classes i and j is $o_{(i-1)n+j}$, where n is the length of the table `seconds` (i.e. its highest index; do not use the Lua length operator `#` here, it will return random values, since the entry at index 1 is always missing for reasons explained just below). Thus, between r (in class f_2) and e (in class s_3), there should be a kern whose width is specified in $o_{(2-1)\times 11+3} = o_{14} = -10$ units — unless, of course, such information is overridden by per-glyph kerning, as seen for *T* above.

When o_i would be 0, i.e. no kerning is specified between two classes, the entry is omitted in the `offsets` table. Another important point is that f_1 and s_1 are special: like class 0 in contextual substitutions above, they hold all the glyphs that don't appear in other classes, and thus aren't explicitly defined. Kerning for those classes is seldom, if ever, specified, for reasons that should be obvious.

LuaTEX is able to handle such kerning by itself; to do so, we fill the `kerns` table that each character may have; e.g. for *r* it would contain the following information:

```
kerns = {
  [name_to_unicode.a]       = -9830,
  [name_to_unicode.aogonek] = -9830,
  [name_to_unicode.c]       = -6554,
  ... }
```

assuming the font has been loaded at 10pt and there are 1000 units per em, so that the first value is $\frac{-15*10*65536}{1000} = 9830.4$ scaled points (rounded, since there is nothing smaller than a scaled point). However, LuaTEX won't insert a kern if anything occurs between two glyph nodes (barring discretionary nodes, which it will certainly do better than us), and we have just seen such intervening material in the previous subsection with capital spacing.[47] Also,

pens quite regularly with similar `lookup` entries. The correct information can be reliably retrieved with:

```
local lk = (type(lookup) == "table" and lookup[1])
           or lookup
```

[46] Actually, `kernclass` is made of subtables, each one being a kerning table; but only the first is valid, the others are defined elsewhere and should be ignored when repeated as the non-first subtables.

[47] Capital spacing is a single positioning lookup. But given that the tag has additional semantics, namely that the positioning should occur if and only if the uppercase letter is preceded and/or followed by another capital letter, we can reinterpret it as kerning with the pairs *A/A*, *A/B* ... *Z/Y*, *Z/Z*. Then, for each glyph m on the left and glyph n on the right, we can set entry n in `kerns`, as $h_m + x_n + k_{mn}$, where

the kerning lookup here is instructed to ignore marks (see the `flags` entry), and LuaTEX can't do that.

Mark positioning. We turn now to the placement of diacritics, i.e. glyphs that exist only relative to another. Only the latter should be visible to the justification engine afterward; the diacritic itself may only contribute to the vertical dimension of the resulting combination.

Since Unicode defines many precomposed characters, independent diacritics can often be avoided. Yet situations still abound where mark positioning is crucial — if only because the font has no glyph for a given precomposed character. As an example, the International Phonetic Alphabet uses a ring below a symbol to denote a voiceless sound (when no independent symbol exists), e.g. *m̥*.[48] If we want to do phonetics with Libertine, we'll have to use mark positioning, because the symbol doesn't exist in that font (or in Unicode, for that matter).

At this point, it might be useful to review how TEX positions diacritics. The `\accent` primitive, used as:

`\accent` ⟨*number*⟩ ⟨*char*⟩

places the character at position ⟨*number*⟩ in the font on the character at position ⟨*char*⟩ as follows: first, if ⟨*char*⟩'s height differs from the font's x-height, i.e. `\fontdimen5` (a.k.a. `parameters.x_height`), then the accent is put into a box shifted vertically by:

$$shift = height_{chr} - xheight$$

Then the accent is surrounded by kerns so that it is centered on the accentee, modulo the declared slant per point (`\fontdimen1` a.k.a. `parameters.slant`). If we denote the width of the first kern by k_1 and the second by k_2, they can be computed as:

$$k_1 = \frac{width_{chr} - width_{acc}}{2} + \frac{slant}{1pt} \times shift$$

$$k_2 = -(k_1 + width_{acc})$$

We can see that the second kern is used only to cancel whatever horizontal displacement was caused by the first kern and the accent. The entire operation can be justified by saying that TEX expects an accent to be designed to fit an ascenderless lowercase letter.

OpenType fonts don't work this way; instead, glyphs have anchors, and a diacritic is placed relative to a base glyph by aligning those anchors. Also, following Unicode, a diacritic is expected to follow the character it modifies: `e^` denotes *ê*. What road shall we follow: TEX's diacritic–base or Unicode's base–diacritic? And does it make any difference?

To see the problem in action, suppose we're scanning a node list to perform mark positioning. If marks were denoted in TEX's way, then they can easily be spotted thanks to the kerns used to position them, which have `subtype` 2. We can undo whatever TEX tried to do and reposition the marks. If we follow Unicode, however, TEX will do nothing and would-be diacritics will simply stand suspended in mid-air with nothing to distinguish them.

But OpenType fonts are a little bit more clever than that: We can rely on glyph classes. Whenever a "mark" glyph is encountered, it should be attached to the preceding glyph (if the corresponding feature has been activated). Thus, when scanning a node list, we should check the class of each glyph node (actually, we've been implicitly doing that all along, since we know some lookups ignore some glyph classes). Not all fonts have classes and anchors, however, only those that implement mark positioning. For the rest (e.g. Latin Modern), one should rely on TEX's `\accent`, because we won't be able to do better than that. But otherwise the OpenType method should be followed because if we put diacritics before their bases *à la* TEX, as in `A`⟨*mark*⟩`BC` where OpenType expects `AB`⟨*mark*⟩`C`, then they might mess with a lookup designed to act on `A` and `B` and not instructed to ignore marks because there shouldn't be one to begin with.

Let's get back to voiceless *m* (109). If we explore the letter itself, we'll see it has an `anchors` subtable (also, not shown here is the glyph's `class`: it is `base`):

```
anchors = {
 basechar = {
  Anchor-2 = { x = 522, y = 645,  lig_index = 0 }
  Anchor-5 = { x = 157, y = 9,    lig_index = 0 }
  Anchor-6 = { x = 382, y = -105, lig_index = 0 }
 } }
```

The anchors here all belong to the same category, `basechar`, meaning that they are the anchors to be used when a mark is to be attached to the letter. The ring-below glyph (755), on the other hand, has the following anchors (and its `class` is `mark`):

```
anchors = {
  mark = { Anchor-6 = { x = 92, y = -116,
                        lig_index = 0 } } }
```

---

h and *x* are the corresponding entries in `specification` for the single positioning lookup and *k* is the (real) kern between the two glyphs. Then we can let LuaTEX do the kerning by itself, although, of course, this is just a special case.

[48] A voiceless consonant is produced with the vocal folds open, so that the air flows through soundlessly; in a voiced consonant, the vocal folds are closed and flap under air pressure, producing a vibration. In each of the pairs *t/d*, *p/b* and *f/v*, both sounds are identical except that the first one is voiceless while the second is voiced. To produce a voiceless *m*, either say *m* without humming, or say *p* while expelling air through the nose, or whisper *mama*, although technically whispering isn't voicelessness.

The only anchor here is in the `mark` category, meaning it should be used when the glyph is moved relative to another. In both cases, the anchor category might seem redundant given the glyph's class, but we'll see below that it is not: there might be anchors with different categories.

Now, if we want to position the ring (the mark) relative to m (the base), we have to align the anchor of category `basechar` in the latter with the corresponding anchor of category `mark` in the former; in this case, the anchor is `Anchor-6`. So, assuming we're doing it the Unicode way, i.e. base followed by mark, we have to align the two glyphs at the origin, which, in a left-to-right writing system, means that we must move the mark left by the base's width, then shift it horizontally by $x_{base} - x_{mark}$ and vertically by $y_{base} - y_{mark}$. Finally, all this movement should be invisible, so that after the operation we end up at the base's right border. We could use kerns and an hbox to do that, but LuaTEX offers a much simpler solution, manipulating the mark's `xoffset` and `yoffset` fields (which are present in all glyph nodes). We still need a kern after the mark to cancel its width, though.

But we've moved a bit too fast. Now that we're more knowledgeable about lookups, we know that we should always check their subtables, because they give us the order of operations; so let's have a look at `mb_l_3` in `gpos` (triggered by the `mark` tag):

```
name = mb_l_3
type = gpos_mark2base
features = { ... }
flags = { }
subtables = {
  1 = { name = mb_l_3_a_0, anchor_classes = 1 }
  2 = { name = mb_l_3_a_1, anchor_classes = 1 }
  ⟨... five more ...⟩
  }
```

(We furtively note the lookup's `type`.) The subtables have names, as usual; however, we haven't seen those names in the glyph's information for this lookup, as was the case for other types. That's because the link between the lookup in `gpos` and its details in affected glyphs is indirect in this case; subtables are tied to anchors (as indicated, somewhat redundantly, by the `anchor_classes` field), which anchors we then find in the glyphs, as we have already seen. Anchors are enumerated in `fonttable.anchor_classes`; here are the first two:

```
1 = { type = mark
      name = Anchor-0
      lookup = { 1 = mb_l_3_a_0, ... } }
2 = { type = mark
      name = Anchor-1
      lookup = { 1 = mb_l_3_a_1, ... } }
```

(Again, `lookup` is buggy; the correct value is the first entry.) Now, to retrace our steps: the `mb_l_3` lookup has subtables, each associated with one or more anchor(s), which anchors we then find in some glyph(s). Thus, to implement mark positioning on a given mark glyph: for each subtable of the lookup, and for each anchor in that subtable, we check if the mark has this anchor in its `anchors.mark` subtable and if the nearest preceding base glyph has this anchor in its `anchors.basechar` subtable. If so, we align the two anchors (and the lookup, as usual, is considered processed: subsequent anchors and subtables are ignored). Now, the reader can check that the seventh subtable in `mb_l_3` points to `Anchor-6`, which is exactly the one we need to, at long last, put that ring below m.

Now suppose we're in a node list, with a node `mark` to be positioned on the glyph `base` immediately on its left. We've retrieved the necessary anchor for each glyph, which we denote with `ma` for the mark's anchor and `ba` for the base's anchor. Then here's how the positioning is to be performed:

```
mark.xoffset = ba.x - ma.x - base.width
mark.yoffset = ba.y - ma.y
local kern = node.new("kern", 2)
kern.kern = -mark.width
head = node.insert_after(head, mark, kern)
```

The kern has subtype 2, as a reminder that it is used for accent placement. Note that `xoffset` is invisible to TEX, so there is no need to take it into account in the kern's width.[49] Also, marks often have no width (i.e. they are drawn entirely to the left of their bounding boxes), in which case the kern may be avoided entirely.[50]

On the other hand, `yoffset` does have an effect, but only on height, not depth; this means that if we're dealing with a mark placed under the baseline, and `yoffset` is non-zero, then the depth of the horizontal box containing the character might not be properly computed. Devising an alternate solution, using kerns and boxes as TEX natively does and computing depth properly, is left as an exercise to the reader.[51]

[49] In right-to-left typesetting, we would have to move the mark right by its width, not the base's width, since glyphs are always drawn with the cartesian origin at the bottom left corner; however, we would still use a negative value for `xoffset`, because this field follows the writing direction.

[50] Actually, no matter what `fonttable` says, we could set the widths of all mark glyphs to 0, given that they are supposed to be non-spacing glyphs. Then we could do without the kern altogether, thus tinkering less with the nodelist.

[51] Said reader may also be glad to learn that LuaTEX has primitives (inherited from PDFTEX) that set the height and depth of paragraph lines independently of their contents:

Mark to mark positioning. Positioning a mark on a base glyph is not the only possibility in diacritic placement; we may want to position a mark, let's call it `mark1`, relative to another one, `mark2`; this occurs for instance if you want a tone marker and an accent on the same vowel in Vietnamese: the first should be placed relative to the second, not relative to the third.[52] The process is similar to what we've just seen, except that: the lookup's type in `fonttable.gpos` is `gpos_mark2mark`, and in `anchor_classes` the type is `mkmk`; `mark2`'s anchor is to be found in the `basemark` subtable of the `anchors` table (parallel to the `basechar` subtable for a base glyph); and most importantly, since `mark2` itself is very likely to be moved on the nearest base glyph, this positioning, reflected in the `xoffset` and/or `yoffset` of `mark2`, should be taken into account when moving `mark1` (actually, we should have done that with mark-to-base placement too, because it may well happen that a base glyph is moved, if only because the end user does so by hand). Libertine contains no lookup of this type.

Mark to ligature positioning. Finally, a mark may be positioned relative to a component in a ligature. Recall the "`c¸h`" example discussed above: the ligaturing lookup (which ignores marks) will turn it into "*ch¸*", and we should position the cedilla so as to obtain "*çh*". This is not simple mark-to-base positioning, because the mark could (theoretically, at least) be set on either of the two original components. That is why an anchor in a ligature has several instances of itself, each associated with one of the original components (not all components need to have an anchor, though); so, if a mark should be placed relative to the first component, the first instance of the anchor shall be used, while for the second component the second instance is the right one, and so on.

Of course, this implies the ligaturing lookup has stored the necessary information, namely the component with which the mark was originally associated. To do so, we can set an attribute in the mark node; in our case, we would set this attribute to 1, indicating that the mark is associated with the first component.

In Libertine, the `mklg` tag activates the `ml_l_4` lookup, whose `type` is `gpos_mark2ligature`.[53] This lookup has one subtable associated with `Anchor-7` (whose type in `anchor_classes` is `mark`, but should be `mklg` — no relation to the tag). If we look at the cedilla (184), we'll see that it has this anchor in the `mark` subtable — so we can at least conclude that, no matter what X is in "mark-to-X positioning", the mark's anchor always has the same category. But the `anchors` table for the ligature (57, 403) is different:

```
anchors = {
  baselig = {
    Anchor-7 = { 1 = { x = 212
                       y = 2
                       lig_index = 0 }
                 2 = { x = 470
                       y = 2
                       lig_index = 1 } } } }
```

Anchors of type `baselig` are made of subtables, each specifying one instance of a given anchor. The index of each subtable correctly identifies the associated component in the ligature, because subtables aren't continuously numbered. If `Anchor-7` was defined for the second component only, the subtable would still be at index 2, not 1. So the `lig_index` field can be done without, all the more as it has the inhuman habit of starting counting at 0.

As for the implementation, it is identical to what we've already seen: anchors should be aligned. The only difference is in how to find the right anchor.

Cursive attachment. In a script (cursive) font, letters should be properly attached together. The Latin alphabet poses no problem: all letters are on the baseline anyway, so it is up to the font designer to ensure that exit points and entry points match, at least vertically (horizontal adjustment can be left to kerning); in other words, entry and exit points should be at the same height. However, that is not true of some Arabic scripts, such as Nasta'līq,[54]

`\pdfeachlineheight` and `\pdfeachlinedepth`. The wrongly computed depth can then be ignored, except when building an independent hbox.

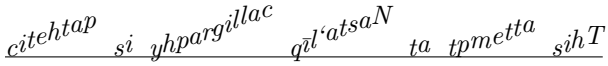
[52] Tone is the use of pitch as a lexical device (i.e. to distinguish between words), just like phonemes do in non-tonal (and, of course, tonal) languages; that is different from intonation, used in all languages, which do not distinguish words: if you say *cat* and then repeat it with a raising intonation, as in a question, it's still the same word.

A little less off-topic: there exist precomposed characters for Vietnamese in Unicode, and they have glyphs in Libertine, so the example is slightly spurious.

[53] Mark to ligature positioning is usually activated with the `mark` tag, just like mark to base positioning. Here I've used a different tag just so users can test it independently.

[54] Nasta'līq is used mostly for Indo-European languages (especially, Urdu), thus totally unrelated (or rather, to this day not convincingly — for most linguists — related) to Arabic or the Semitic languages more generally or even the Afro-Asiatic family, unless one is willing to accept the Nostratic superfamily or even more remote and controversial linguistic classifications. But then, languages as unrelated to Latin as Basque, Mohawk, Vietnamese, Wolof, and many, many more, are written in the Latin alphabet.

where entry points are generally higher than exit points, and where only the last glyph of a word is set on the baseline:



The position of each glyph thus depends on the position of the next in the word. To implement that, anchors are used again, but this time each anchor is twofold: there is an entry point and an exit point. For each pair of consecutive glyphs, the entry point of the second glyph should be aligned with the exit point of the first glyph, and a kern and `yoffset` are all we need to do so. Although the operation is similar to mark positioning, it differs in one important respect: all glyphs are spacing here, so there is no kern to compensate for the second glyph's width. Libertine has no cursive positioning, but a typical `anchors` table in a glyph would look like:

```
anchors = {
  centry = {
    Cursive-8 = { x = ⟨x1⟩
                  y = ⟨y1⟩
                  lig_index = 0 } }
  cexit = {
    Cursive-8 = { x = ⟨x2⟩
                  y = ⟨y2⟩
                  lig_index = 0 } }
   ⟨other anchors⟩
  }
```

Thus, anchors for cursive attachment are identical to other anchors, except that they belong to the `centry` and `cexit` subtables. As for the lookup itself in `gpos`, it has the `gpos_cursive` type, whereas the anchors' type in `anchor_classes` is `curs`.

As mentioned above, the lookup is mostly used to attach glyphs in words with the last glyph on the baseline; that means that positioning should begin at the end of the word and progress contrary to the direction of writing. This stipulation is not part of the lookup type itself; instead, a flag (in the `flags` subtable of the lookup in `gpos`) is used: `r2l`, meaning "right-to-left".[55] This in turn implies that we have a definition of a "word"; space will do, as discussed a few pages ago.

(Chaining) contextual positioning. We'll omit discussion of these two lookup types, not because there is nothing interesting here, but simply because they are identical to (chaining) contextual substitutions, except that they dispatch to positioning, not substitution, lookups. I hope the reader has the buoyant feeling associated with the last-minute cancellation of a dreaded two-hour class (each Friday at 5 PM). But let's just say that contextual positioning can be used e.g. in *Wörter* (German for *words*) to lower the umlaut so that kerning can be increased with the preceding *W* (this of course requires that the *o* and the accent are separate glyphs, probably due to a multiple substitution), or to add kerning between the period and *T* in *S.A.T.*, the kerning being actually (visually) with the preceding *A*.

13 Conclusion

I hope the reader has found the foregoing journey into the world of OpenType fonts and LuaTEX interesting, informative, and enticing. A larger world lies beyond, especially regarding non-Latin writing systems, not to mention maths, and I'll be satisfied if the reader now feels confident enough to step into that world. As with all of LuaTEX, it may seem intimidating at first but is ultimately extremely rewarding.

References

[1] Yannis Haralambous. *Fonts & Encodings*. O'Reilly, 2007. First edition in French as *Fontes & Codages*, O'Reilly, 2004.

[2] Taco Hoekwater. Math in LuaTEX 0.40. *MAPS*, 38, 2009. An updated version was translated in French as "LuaTEX 0.65 et les mathématiques" in *Cahiers Gutenberg*, 54–55, 2010.

[3] Paul Isambert. LuaTEX: What it takes to make a paragraph. *TUGboat* 32:1, 2011. `tug.org/TUGboat/tb32-1/tb100isambert.pdf`.

[4] LuaTEX team. *LuaTEX Reference*. `www.luatex.org/svn/trunk/manual/luatexref-t.pdf`.

[5] Microsoft OpenType specification. `www.microsoft.com/typography/otspec`.

[6] Linux Libertine. `www.linuxlibertine.org`.

[7] Ulrik Vieth. OpenType math illuminated. *TUGboat* 30:1, 2009. `tug.org/TUGboat/tb30-1/tb94vieth.pdf`.

[8] George Williams. FontForge documentation. `fontforge.sourceforge.net`.

◇ Paul Isambert
`zappathustra (at) free dot fr`

[55] Although cursive attachment is mostly used in a right-to-left writing system, the flag's name (inherited from OpenType files, not made up by LuaTEX or FontForge) assumes a left-to-right system, since it means "contrary to the writing direction".

ConTEXt: Updating the code base

Hans Hagen

1 Introduction

After much experimenting with new code in MkIV a new stage in ConTEXt development was entered in the last quarter of 2011. This was triggered by several more or less independent developments. I will discuss some of them here since they are a nice illustration of how ConTEXt evolves.

2 Interfacing

Wolfgang Schuster, Aditya Mahajan and I were experimenting with an abstraction layer for module writers. In fact this layer itself was a variant of some new mechanisms used in the MkIV structure related code. That code was among the first to be adapted as it is accompanied by much Lua code and has been performing rather well for some years now.

In ConTEXt most of the user interface is rather similar and module writers are supposed to follow the same route as the core of ConTEXt. For those who have looked in the source the following code might look familiar:

```
\unexpanded\def\mysetupcommand
  {\dosingleempty\domysetupcommand}

\def\domysetupcommand[#1]%
  {..........
   \getparameters[\??my][#1]%
   ..........}
```

This implements the command `\mysetupcommand` that is used as follows:

```
\mysetupcommand[color=red,style=bold,...]
```

The above definition uses three rather low-level interfacing commands. The `\unexpanded` makes sure that the command does not expand in unexpected ways in cases where expansion is less desirable. (Aside: The ConTEXt `\unexpanded` prefix has a long history and originally resulted in the indirect definition of a macro. That way the macro could be part of testing (expanded) equivalence. When ε-TEX functionality showed up we could use `\protected` but we stuck to the name `\unexpanded`. So, currently ConTEXt's `\unexpanded` is equivalent to ε-TEX's `\protected`. Furthermore, in ConTEXt `\expanded` is not the same as the ε-TEX primitive. In order to use the primitives you need to use their `\normal...` synonyms.) The `\dosingleempty` makes sure that one argument gets seen by injecting a dummy when needed. At some point the `\getparameters` command will store the values of keys in a namespace that is determined by `\??my`. The namespace used here is actually one of the internal namespaces which can be deduced from the double question marks. Module namespaces have four question marks.

There is some magic involved in storing the values. For instance, keys are translated from the interface language into the internal language which happens to be English. This translation is needed because a new command is generated:

```
\def\@@mycolor{red}
\def\@@mystyle{bold}
```

and such a command can be used internally because in so-called unprotected mode `@?!` are valid in names. The Dutch equivalent is:

```
\mijnsetupcommando[kleur=rood,letter=vet]
```

and here the `kleur` has to be converted into `color` before the macro is constructed. Of course values themselves can stay as they are as long as checking them uses the internal symbolic names that have the language specific meaning.

```
\c!style{color}
\k!style{kleur}
\v!bold {vet}
```

Internally assignments are done with the `\c!` variant, translation of the key is done using the `\k!` alternative and values are prefixed by `\v!`.

It will be clear that for the English user interface no translation is needed and as a result that interface is somewhat faster. There we only need

```
\c!style{color}
\v!bold {bold}
```

Users never see these prefixed versions, unless they want to define an internationalized style, in which case the form

```
\mysetupcommand[\c!style=\v!bold]
```

has to be used, as it will adapt itself to the user interface. This leaves the `\??my` that in fact expands to `\@@my`. This is the namespace prefix.

Is this the whole story? Of course it isn't, as in ConTEXt we often have a generic instance from which we can clone specific alternatives; in practice, the `\@@mycolor` variant is used in a few cases only. In that case a setup command can look like:

```
\mysetupcommand[myinstance][style=bold]
```

And access to the parameters is done with:

```
\getvalue{\??my myinstance\c!color}
```

So far the description holds for MkII as well as MkIV, but in MkIV we are moving to a variant of this. At the cost of a bit more runtime and helper macros, we can get cleaner low-level code. The magic word here is `commandhandler`. At some point the new MkIV code started using an extra abstraction layer, but the code needed looked rather repetitive despite

subtle differences. Then Wolfgang suggested that we should wrap part of that functionality in a definition macro that could be used to define module setup and definition code in one go, thereby providing a level of abstraction that hides some nasty details. The main reason why code could look cleaner is that the experimental core code provided a nicer inheritance model for derived instances and Wolfgang's letter module uses that extensively. After doing some performance tests with the code we decided that indeed such an initializer made sense. Of course, after that we played with it, some more tricks were added, and eventually I decided to replace the similar code in the core as well, that is: use the installer instead of defining helpers locally.

So, how does one install a new setup mechanism? We stick to the core code and leave modules aside for the moment.

```
\definesystemvariable{my}
\installcommandhandler \??my {whatever} \??my
```

After this command we have available some new helper commands of which only a few are mentioned here (after all, this mechanism is still somewhat experimental):

```
\setupwhatever[key=value]
\setupwhatever[instance][key=value]
```

Now a value is fetched using a helper:

```
\namedwhateverparameter{instance}{key}
```

However, more interesting is this one:

```
\whateverparameter{key}
```

For this to work, we need to set the instance:

```
\def\currentwhatever{instance}
```

Such a current state macro already was used in many places, so it fits into the existing code quite well. In addition to `\setupwhatever` and friends, another command becomes available:

```
\definewhatever[instance]
\definewhatever[instance][key=value]
```

Again, this is not so much a revolution as we can define such a command easily with helpers, but it pairs nicely with the setup command. One of the goodies is that it provides the following feature for free:

```
\definewhatever[instance][otherinstance]
\definewhatever[instance][otherinstance][key=val]
```

In some cases this creates more overhead than needed because not all commands have instances. On the other hand, some commands that didn't have instances yet, now suddenly have them. For cases where this is not needed, we provide simple variants of commandhandlers.

Additional commands can be hooked into a setup or definition so that for instance the current situation can be updated or extra commands can be defined for this instance, such as `\start...` and `\stop...` commands.

It should be stressed that the installer itself is not that special in the sense that we could do without it, but it saves some coding. More important is that we no longer have the `@@` prefixed containers but use `\whateverparameter` commands instead. This is definitely slower than the direct macro, but as we often deal with instances, it's not that much slower than `\getvalue` and critical components are rather well speed-optimized anyway.

There is, however, a slowdown due to the way inheritance is implemented. That is how this started out: using a different (but mostly compatible) inheritance model. In the MkII approach (which is okay in itself) inheritance happens by letting values point to the parent value. In the new model we have a more dynamic chain. It saves us macros but can expand quite wildly depending on the depth of inheritance. For instance, in sectioning there can easily be five or more levels of inheritance. So, there we get slower processing. The same is true for `\framed` which is a rather critical command, but there it is nicely compensated by less copying. My personal impression is that due to the way ConTEXt is set up, the new mechanism is actually more efficient on an average job. Also, because many constructs also depend on the `\framed` command, that one can easily be part of the chain, which again speeds up a bit. In any case, the new mechanisms use much less hash space.

Some mechanisms still look too complex, especially when they hook into others. Multiple inheritance is not trivial to deal with, not only because the meaning of keys can clash, but also because supporting it would demand quite complex fully expandable resolvers. So for the moment we stay away from it. In case you wonder why we cannot delegate more to Lua: it's close to impossible to deal with TEX's grouping in efficient ways at the Lua end, and without grouping available TEX becomes less useful.

Back to the namespace. We already had a special one for modules but after many years of ConTEXt development, we started to run out of two character combinations and many of them had no relation to what name they spaced. As the code base is being overhauled anyway, it makes sense to also provide a new core namespace mechanism. Again, this is nothing revolutionary but it reads much more nicely.

```
\installcorenamespace {whatever}
\installcommandhandler
   \??whatever {whatever} \??whatever
```

This time deep down no `@@` is used, but rather something more obscure. In any case, no one will use

the meaning of the namespace variables, as all access to parameters happens indirectly. And of course there is no speed penalty involved; in fact, we are more efficient. One reason is that we often used the prefix as follows:

```
\setvalue{\??my:option:bla}{foo}
```

and now we just say:

```
\installcorenamespace {whateveroption}
\setvalue{\??whateveroption bla}{foo}
```

The commandhandler does such assignments slightly differently as it has to prevent clashes between instances and keywords. A nice example of such a clash is this:

```
\setvalue{\??whateveroption sectionnumber}{yes}
```

In sectioning we have instances named `section`, but we also have keys named `number` and `sectionnumber`. So, we end up with something like this:

```
\setvalue
  {\??whateveroption section:sectionnumber}{yes}
\setvalue
  {\??whateveroption section:number}{yes}
\setvalue{\??whateveroption :number}{yes}
```

When I decided to replace code similar to that generated by the installer a new rewrite stage was entered. Therefore one reason for explaining this here is that in the process of adapting the core code instabilities are introduced and as most users use the beta version of MkIV, some tolerance and flexibility is needed and it might help to know why something suddenly fails.

In itself using the commandhandler is not that problematic, but wherever I decide to use it, I also clean up the related code and that is where the typos creep in. Fortunately Wolfgang keeps an eye on the changes so problems that users report on the mailing lists are nailed down relatively fast. Anyway, the rewrite itself is triggered by another event but that one is discussed in the next section.

We don't backport (low-level) improvements and speedups to MkII, because for what we need TEX for, we consider pdfTEX and XETEX rather obsolete. Recent tests show that at the moment of this writing a LuaTEX MkIV run is often faster than a comparable pdfTEX MkII run (using UTF-8 and complex font setups). When compared to a XETEX MkII run, a LuaTEX MkIV run is often faster, but it's hard to compare, as we have advanced functionality in MkIV that is not (or differently) available in MkII.

3 Lexing

The editor that I use, called SciTE, has recently been extended with an extra external lexer module that makes more advanced syntax highlighting possible, using the Lua LPEG library. It is no secret that the user interface of ConTEXt is also determined by the way structure, definitions and setups can be highlighted in an editor.[1] When I changed to SciTE I made sure that we had proper highlighting there.

At Pragma one of the leading principles has always been: if the document source looks bad, mistakes are more easily made and the rendering will also be affected. Or phrased differently: if we cannot make the source look nice, the content is probably not structured that well either. The same is true for TEX source, although to a large extent there one must deal with the specific properties of the language.

So, syntax highlighting, or more impressively: lexing, has always been part of the development of ConTEXt and for instance the pretty printers of verbatim provide similar features. For a long time we assumed line-based lexing, mostly for reasons of speed. And surprisingly, that works out quite well with TEX. We used a simple color scheme suitable for everyday usage, with not too intrusive coloring. Of course we made sure that we had runtime spell checking integrated, and that the different user interfaces were served well.

But then came the LPEG lexer. Suddenly we could do much more advanced highlighting. Once I started playing with it, a new color scheme was set up and more sophisticated lexing was applied. Just to mention a few properties:

- We distinguish between several classes of macro names: primitives, helpers, interfacing, and user macros.
- In addition we highlight constant values and special registers differently.
- Conditional constructs can be recognized and are treated as in any regular language (keep in mind that users can define their own).
- Embedded MetaPost code is lexed independently using a lexer that knows the language's primitives, helpers, user macros, constants and of course specific syntax and drawing operators. Related commands at the TEX end (for defining and processing graphics) are also dealt with.
- Embedded Lua is lexed independently using a lexer that not only deals with the language but also knows a bit about how it is used in ConTEXt. Of course the macros that trigger Lua code are handled.
- Metastructure and metadata related macros are colored in a fashion similar to constants (after

[1] It all started with `wdt`, `texedit` and `texwork`, editors and environments written by myself in Modula2 and later in Perl Tk, but that was in a previous century.

all, in a document one will not see any constants, so there is no color clash).
- Some special and often invisible characters get a special background color so that we can see when there are for instance non-breakable spaces sitting there.
- Real-time spell checking is part of the deal and can optionally be turned on. There we distinguish between unknown words, known but potentially misspelled words, and known words.

Of course we also made lexers for MetaPost, Lua, XML, PDF and text documents so that we have a consistent look and feel.

When writing the new lexer code, and testing it on sources, I automatically started adapting the source to the new lexing where possible. Actually, as cleaning up code is somewhat boring, the new lexer is adding some fun to it. I'm not so sure if I would have started a similar overhaul so easily otherwise, especially because the rewrite now also includes speedup and cleanup. At least it helps to recognize less desirable left-overs of MkII code.

4 Hiding

It is interesting to notice that users seldom define commands that clash with low level commands. This is of course a side effect of the fact that one seldom needs to define a command, but nevertheless. Low-level commands were protected by prefixing them by one or more (combinations of) `do`, `re` and `no`'s. This habit is a direct effect of the early days of writing macros. For TEX it does not matter how long a name is, as internally it becomes a pointer anyway, but memory consumption of editors, loading time of a format, string space and similar factors determined the way one codes in TEX for quite a while. Nowadays there are hardly any limits and the stress that ConTEXt puts on the TEX engine is even less than in MkII as we delegate many tasks to Lua. Memory comes cheap, editors can deal with large amount of data (keep in mind that the larger the file gets, the more lexing power can be needed), and screens are wide enough not to lose part of long names in the edges.

Another development has been that in LuaTEX we have lots of registers so that we no longer have to share temporary variables and such. The rewrite is a good moment to get rid of that restriction.

This all means that at some point it was decided to start using longer command names internally and permit `_` in names. As I was never a fan of using `@` for this, underscore made sense. We have been discussing the use of colons, which is also nice, but has the disadvantage that colons are also used in the source, for instance to create a sub-namespace. When we have replaced all old namespaces, colons might show up in command names, so another renaming roundup can happen.

One reason for mentioning this is that users get to see these names as part of error messages. An example of a name is:

```
\page_layouts_this_or_that
```

The first part of the name is the category of macros and in most cases is the same as the first part of the filename. The second part is a namespace. The rest of the name can differ but we're approaching some consistency in this.

In addition we have prefixed names, where prefixes are used as consistently as possible:

`t_` token register
`d_` dimension register
`s_` skip register
`u_` muskip register
`c_` counter register, constant or conditional
`m_` (temporary) macro
`p_` (temporary) parameter expansion (value of key)
`f_` fractions

This is not that different from other prefixing in ConTEXt apart from the fact that from now on those variables (registers) are no longer accessible in a regular run. We might decide on another scheme but renaming can easily be scripted. In the process some of the old prefixes are being removed. The main reason for changing to this naming scheme is that it is more convenient to grep for them.

In the process most traditional `\if`s get replaced by 'conditionals'. The same is true for `\chardef`s that store states; these become 'constants'.

5 Status

We always try to keep the user interface constant, so most functionality and control stays stable. However, now that most users use MkIV, commands that no longer make sense are removed. An interesting observation is that some users report that low-level macros or registers are no longer accessible. Fortunately that is no big deal as we point them to the official ways to deal with matters. It is also a good opportunity for users to clean up accumulated hackery.

The systematic (file by file) cleanup started in the second half of 2011 and as of January 2012 one third of the core (TEX) modules have to be cleaned up and the planning is to get most of that done as soon as possible. However, some modules will be rewritten (or replaced) and that takes more time. In any case we hope that rather soon most of the code is stable enough that we can start working on new

mechanisms and features. Before that a cleanup of the Lua code is planned.

Although in many cases there are no fundamental changes in the user interface and functionality, I will wrap up some issues that are currently being dealt with. This is just a snapshot of what is happening currently and as a consequence it describes what users can run into due to newly introduced bugs.

The core modules of ConTEXt are loosely organized in groups. Over time there has been some reorganization and in MkIV some code has been moved into new categories. The alphabetical order does not reflect the loading order or dependency tree as categories are loaded intermixed. Therefore the order below is somewhat arbitrary and does not express importance. Each category has multiple files.

5.1 anch: anchoring and positioning

More than a decade ago we started experimenting with position tracking. The ability to store positional information and use that in a second pass permits for instance adding backgrounds. As this code interacts nicely with (runtime) MetaPost it has always been quite powerful and flexible on the one hand, but at the same time it was demanding in terms of runtime and resources. However, were it not for this feature, we would probably not be using TEX at all, as backgrounds and special relative positioning are needed in nearly all our projects.

In MkIV this mechanism had already been ported to a hybrid form, but recently much of the code has been overhauled and its MkII artifacts stripped. As a consequence the overhead in terms of memory probably has increased but the impact on runtime has been considerably reduced. It will probably take some time to become stable if only because the glue to MetaPost has changed. There are some new goodies, like backgrounds behind parshapes, something that probably no one uses and is always somewhat tricky but it was not too hard to support. Also, local background support has been improved which means that it's easier to get them in more column-based layouts, several table mechanisms, floats and such. This was always possible but is now more automatic and hopefully more intuitive.

5.2 attr: attributes

We use attributes (properties of nodes) a lot. The framework for this had been laid early in MkIV development, so not much has changed here. Of course the code gets cleaner and hopefully better as it is putting quite a load on the processing. Each new feature depending on attributes adds some extra overhead even if we make sure that mechanisms only kick in when they are used. This is due to the fact that attributes are linked lists and although unique lists are shared, they travel with each node. On the other hand, the cleanup (and de-MkII-ing) of code leads to better performance so on the average no user will notice this.

5.3 back: backend code generation

This category wraps backend issues in an abstract way that is similar to the special drivers in MkII. So far we have only three backends: PDF, XML, and XHTML. Such code is always in a state of maintenance, if only because backends evolve.

5.4 bibl: bibliographies

For a while now, bibliographies have not been an add-on but part of the core. There are two variants: traditional BIBTEX support derived from a module by Taco Hoekwater but using MkIV features (the module hooks into core code), and a variant that delegates most work to Lua by creating an in-memory XML tree that gets manipulated. At some point I will extend the second variant. Going the XML route also connects better with developments such as Jean-Michel Hufflen's MlBIBTEX.

5.5 blob: typesetting in Lua

Currently we only ship a few helpers but eventually this will become a framework for typesetting raw text in Lua. This might be handy for some projects that we have where the only input is XML, but I'm not that sure if it will produce nice results and if the code will look better. On the other hand, there are some cases where in a regular TEX run some basic typesetting in Lua might make sense. Of course I also need an occasional pet project so this might qualify as one.

5.6 buff: buffers and verbatim

Traditionally buffers and verbatim have always been relatives as they share code. The code was among the first to be adapted to LuaTEX. There is not that much to gain in adapting it further. Maybe I will provide more lexers for pretty-printing some day.

5.7 catc: catcodes

Catcodes are a rather TEX-specific feature and we have organized them in catcode regimes. The most important recent change has been that some of the characters with a special meaning in TEX (like ampersand, underscore, superscript, etc.) are no longer special except in cases that matter. This somewhat incompatible change surprisingly didn't lead to many problems. Some code that is specific for the MkII

XML processor has been removed as we no longer assume it is being used in MkIV.

5.8 char: characters

This important category deals with characters and their properties. Already from the beginning of MkIV character properties have been (re)organized in Lua tables and therefore much code deals with it. The code is rather stable but occasionally the tables are updated as they depend on developments in Unicode. In order to share as much data as possible and prevent duplicates there are several inheritance mechanisms in place but their overhead is negligible.

5.9 chem: chemistry

The external module that deals with typesetting chemistry was transformed into a MkIV core module some time ago. Not much has changed in this department but some enhancements are pending.

5.10 cldf: ConTEXt Lua documents

These modules are mostly Lua code and are the interface into ConTEXt as well as providing ways to code complete documents in Lua. This is one of those categories that is visited every now and then to be adapted to improvements in other core code or in LuaTEX. This is one of my favourite categories as it exposes most of ConTEXt at the Lua end which permits writing solutions in Lua while still using the full power of ConTEXt. A dedicated manual is on its way.

5.11 colo: colors and transparencies

This is rather old code, and apart from some cleanup not much has been changed here. Some macros that were seldom used have been removed. One issue that is still pending is a better interface to MetaPost as it has different color models and we have adapted code at that end. This has a rather low priority because in practice it is no real problem.

5.12 cont: runtime code

These modules contain code that is loaded at runtime, such as filename remapping, patches, etc. It does not make much sense to improve these.

5.13 core: all kinds of core code

Housekeeping is the main target of these modules. There are still some typesetting-related components here but these will move to other categories. This code is cleaned up when there is a need for it. Think of managing files, document project structure, module loading, environments, multipass data, etc.

5.14 data: file and data management

This category hosts only Lua code and hasn't been touched for a while. Here we deal with locating files, caching, accessing remote data, resources, environments, and the like.

5.15 enco: encodings

Because (font) encodings are gone, there is only one file in this category and that one deals with weird (composed or otherwise special) symbols. It also provides a few traditional TEX macros that users expect to be present, for instance to put accents over characters.

5.16 file: files

There is some overlap between this category and core modules. Loading files is always somewhat special in TEX as there is the TEX directory structure to deal with. Sometimes you want to use files in the so-called tree, but other times you don't. This category provides some management code for (selective) loading of document files, modules and resources. Most of the code works with accompanying Lua code and has not been touched for years, apart from some weeding and low-level renaming. The project structure code has mostly been moved to Lua and this mechanism is now more restrictive in the sense that one cannot misuse products and components in unpredictable ways. This change permits better automatic loading of cross references in related documents.

5.17 font: fonts

Without proper font support a macro package is rather useless. Of course we do support the popular font formats but nowadays that's mostly delegated to Lua code. What remains at the TEX end is code that loads and triggers a combination of fonts efficiently. Of course in the process text and math each need to get the proper amount of attention.

There is no longer shared code between MkII and MkIV. Both already had rather different low-level solutions, but recently with MkIV we went a step further. Of course it made sense to kick out commands that were only used for pdfTEX Type 1 and XETEX OpenType support but more important was the decision to change the way design sizes are supported.

In ConTEXt we have basic font definition and loading code and that hasn't conceptually changed much over the years. In addition to that we have so-called bodyfont environments and these have been made a bit more powerful in recent MkIV. Then there are typefaces, which are abstract combinations of fonts and defining them happens in typescripts.

This layered approach is rather flexible, and was greatly needed when we had all those font encodings (to be used in all kinds of combinations within one document). In MkIV, however, we already had fewer typescripts as font encodings are gone (also for Type 1 fonts). However, there remained a rather large blob of definition code dealing with Latin Modern; large because it comes in design sizes.

As we always fall back on Latin Modern, and because we don't preload fonts, there is some overhead involved in resolving design size related issues and definitions. But, it happens that this is the only font that ships with many files related to different design sizes. In practice no user will change the defaults. So, although the regular font mechanism still provides flexible ways to define font file combinations per bodyfont size, resolving to the right best matching size now happens automatically via a so-called Lua font goodie file which brings down the number of definitions considerably. The consequence is that ConTEXt starts up faster, not only in the case of Latin Modern being used, but also when other designs are in play. The main reason for this is that we don't have to parse those large typescripts anymore, as the presets were always part of the core set of typescripts. At the same time loading a specific predefined set has been automated and optimized. Of course on a run of 30 seconds this is not that noticeable, but it is on a 5 second run or when testing something in the editor that takes less than a second. It also makes a difference in automated workflows; for instance at Pragma we run unattended typesetting flows that need to run as fast as possible. Also, in virtual machines using network shares, the fewer files consulted the better.

Because math support was already based on OpenType, where ConTEXt turns Type 1 fonts into OpenType at runtime, nothing fundamental has changed here, apart from some speedups (at the cost of some extra memory). Where the overhead of math font switching in MkII is definitely a factor, in MkIV it is close to negligible, even if we mix regular, bold, and bidirectional math, which we have done for a while.

The low-level code has been simplified a bit further by making a better distinction between the larger sizes (`a` up to `d`) and smaller sizes (`x` and `xx`). These now operate independently of each other (i.e. one can now have a smaller relative `x` size of a larger one). This goes at the cost of more resources but it is worth the effort.

By splitting up the large basic font module into smaller ones, I hope that it can be maintained more easily although someone familiar with the older code will only recognize bits and pieces. This is partly due to the fact that font code is highly optimized.

5.18 grph: graphic (and widget) inclusion

Graphics inclusion is always work in progress as new formats have to be dealt with or users want additional conversions to be done. This code will be cleaned up later this year. The plug-in mechanisms will be extended (examples of existing plug-ins are automatic converters and barcode generation).

5.19 hand: special font handling

As we treat protrusion and hz as features of a font, there is not much left in this category apart from some fine-tuning. So, not much has happened here and eventually the left-overs in this category might be merged with the font modules.

5.20 java: JavaScript in PDF

This code already has been cleaned up a while ago, when moving to MkIV, but we occasionally need to check and patch due to issues with JavaScript engines in viewers.

5.21 lang: languages and labels

There is not much changed in this department, apart from additional labels. The way inheritance works in languages differs too much from other inheritance code so we keep what we have here. Label definitions have been moved to Lua tables from which labels at the TEX end are defined that can then be overloaded locally. Of course the basic interface has not changed as this is typically code that users will use in styles.

5.22 luat: housekeeping

This is mostly Lua code needed to get the basic components and libraries in place. While the `data` category implements the connection to the outside world, this category runs on top of that and feeds the TEX machinery. For instance conversion of MkVI files happens here. These files are seldom touched but might need an update some time (read: prune obsolete code).

5.23 lpdf: PDF backend

Here we implement all kinds of PDF backend features. Most are abstracted via the backend interface. So, for instance, colors are done with a high level command that goes via the backend interface to the `lpdf` code. In fact, there is more such code than in (for instance) the MkII special drivers, but readability comes at a price. This category is always work in progress as insights evolve and users demand more.

5.24 lxml: XML and lpath

As this category is used by some power users we cannot change too much here, apart from speedups and extensions. It's also the bit of code we use frequently at Pragma, and as we often have to deal with rather crappy XML I expect to move some more helpers into the code. The latest greatest trickery related to proper typesetting can be seen in the documents made by Thomas Schmitz. I wonder if I'd still have fun doing our projects if I hadn't, in an early stage of MkIV, written the XML parser and expression parser used for filtering.

5.25 math: mathematics

Math deserves its own category but compared to MkII there is much less code, thanks to Unicode. Since we support Type 1 as virtual OpenType nothing special is needed there (and eventually there will be proper fonts anyway). When rewriting code I try to stay away from hacks, which is sometimes possible by using Lua but it comes with a slight speed penalty. Much of the Unicode math-related font code is already rather old but occasionally we add new features. For instance, because OpenType has no italic correction we provide an alternative (mostly automated) solution.

On the agenda is more structural math encoding (maybe like openmath) but tagging is already part of the code so we get a reasonable export. Not that someone is waiting for it, but it's there for those who want it. Most math-related character properties are part of the character database which gets extended on demand. Of course we keep MathML up-to-date because we need it in a few projects.

We're not in a hurry here but this is something where Aditya and I have to redo some of the code that provides AMS-like math commands (but as we have them configurable some work is needed to keep compatibility). In the process it's interesting to run into probably never-used code, so we just remove those artifacts.

5.26 meta: metapost interfacing

This and the next category deal with MetaPost. This first category is quite old but already adapted to the new situation. Sometimes we add extra functionality but the last few years the situation has become rather stable with the exception of backgrounds, because these have been overhauled completely.

5.27 mlib: metapost library

Apart from some obscure macros that provide the interface between front- and backend this is mostly Lua code that controls the embedded MetaPost library. So, here we deal with extensions (color, shading, images, text, etc.) as well as runtime management because sometimes two runs are needed to get a graphic right. Some time ago, the MkII-like extension interface was dropped in favor of one more natural to the library and MetaPost 2. As this code is used on a daily basis it is quite well debugged and the performance is pretty good too.

5.28 mult: multi-lingual user interface

Even if most users use the English user interface, we keep the other ones around as they're part of the trademark. Commands, keys, constants, messages and the like are now managed with Lua tables. Also, some of the tricky remapping code has been stripped because the setup definitions files are dealt with. These are XML files that describe the user interface that get typeset and shipped with ConTEXt.

These files are being adapted. First of all the commandhandler code is defined here. As we use a new namespace model now, most of these namespaces are defined in the files where they are used. This is possible because they are more verbose so conflicts are less likely (also, some checking is done to prevent reuse). Originally the namespace prefixes were defined in this category but eventually all that code will be gone. This is a typical example where 15-year-old constraints are no longer an issue and better code can be used.

5.29 node: nodes

This is a somewhat strange category as all typeset material in TEX becomes nodes so this deals with everything. One reason for this category is that new functionality often starts here and is sometimes shared between several mechanisms. So, for the moment we keep this category. Think of special kerning, insert management, low-level referencing (layer between user code and backend code) and all kinds of rule and displacement features. Some of this functionality is described in previously published documents.

5.30 norm: normalize primitives

We used to initialize the primitives here (because LuaTEX starts out blank). But after moving that code this category only has one definition left and that one will go too. In MkII these files are still used (and actually generated by MkIV).

5.31 pack: wrapping content in packages

This is quite an important category as in ConTEXt lots of things get packed. The best example is

\framed and this macro has been maximally optimized, which is not that trivial since much can be configured. The code has been adapted to work well with the new commandhandler code and in future versions it might use the commandhandler directly. This is however not that trivial because hooking a setup of a command into \framed can conflict with the two commands using keys for different matters.

Layers are also in this category and they probably will be further optimized. Reimplementing reusable objects is on the horizon, but for that we need a more abstract Lua interface, so that will come first. This has a low priority because it all works well. This category also hosts some helpers for the page builder but the builder itself has a separate category.

5.32 page: pages and output routines

Here we have an old category: output routines (trying to make a page), page building, page imposition and shipout, single and multi column handling, very special page construction, line numbering, and of course setting up pages and layouts. All this code is being redone stepwise and stripped of old hacks. This is a cumbersome process as these are core components where side effects are sometimes hard to trace because mechanisms (and user demands) can interfere. Expect some changes for the good here.

5.33 phys: physics

As we have a category for chemistry it made sense to have one for physics and here is where the unit module's code ended up. So, from now on units are integrated into the core. We took the opportunity to rewrite most of it from scratch, providing a bit more control.

5.34 prop: properties

The best-known property in TEX is a font and color is a close second. Both have their own category of files. In MkII additional properties like backend layers and special rendering of text were supported in this category but in MkIV properties as a generic feature are gone and replaced by more specific implementations in the attr namespace. We do issue a warning when any of the old methods are used.

5.35 regi: input encodings

We still support input encoding regimes but hardly any TEX code is involved now. Only when users demand more functionality does this code get extended. For instant, recently a user wanted a conversion function for going from UTF-8 to an encoding that another program wanted to see.

5.36 scrn: interactivity and widgets

All modules in this category have been overhauled. On the one hand we lifted some constraints, for instance the delayed initialization of fields no longer makes sense as we have a more dynamic variable resolver now (which is somewhat slower but still acceptable). On the other hand some nice but hard to maintain features have been simplified (not that anyone will notice as they were rather special). The reason for this is that vaguely documented PDF features tend to change over time which does not help portability. Of course there have also been some extensions, and it is actually less hassle (but still no fun) to deal with such messy backend related code in Lua.

5.37 scrp: script-specific tweaks

These are script-specific Lua files that help with getting better results for scripts like CJK. Occasionally I look at them but how they evolve depends on usage. I have some very experimental files that are not in the distribution.

5.38 sort: sorting

As sorting is delegated to Lua there is not much TEX code here. The Lua code occasionally gets improved if only because users have demands. For instance, sorting Korean was an interesting exercise, as was dealing with multiple languages in one index. Because sorting can happen on a combination of Unicode, case, shape, components, etc. the sorting mechanism is one of the more complex subsystems.

5.39 spac: spacing

This important set of modules is responsible for vertical spacing, strut management, justification, grid snapping, and all else that relates to spacing and alignments. Already in an early stage vertical spacing was mostly delegated to Lua so there we're only talking of cleaning up now. Although ... I'm still not satisfied with the vertical spacing solution because it is somewhat demanding and an awkward mix of TEX and Lua which is mostly due to the fact that we cannot evaluate TEX code in Lua.

Horizontal spacing can be quite demanding when it comes down to configuration: think of a table with 1000 cells where each cell has to be set up (justification, tolerance, spacing, protrusion, etc.). Recently a more drastic optimization has been done which permits even more options but at the same time is much more efficient, although not in terms of memory.

Other code, for instance spread-related status information, special spacing characters, interline spacing and linewise typesetting all falls into this category

and there is probably room for improvement there. It's good to mention that in the process of the current cleanup hardly any Lua code gets touched, so that's another effort.

5.40 strc: structure

Big things happened here but mostly at the TEX end as the support code in Lua was already in place. In this category we collect all code that gets or can get numbered, moves around and provides visual structure. So, here we find itemize, descriptions, notes, sectioning, marks, block moves, etc. This means that the code here interacts with nearly all other mechanisms.

Itemization now uses the new inheritance code instead of its own specific mechanism but that is not a fundamental change. More important is that code has been moved around, stripped, and slightly extended. For instance, we had introduced proper `\startitem` and `\stopitem` commands which are somewhat conflicting with `\item` where a next instance ends a previous one. The code is still not nice, partly due to the number of options. The code is a bit more efficient now but functionally the same.

The sectioning code is under reconstruction as is the code that builds lists. The intention is to have a better pluggable model and so far it looks promising. As similar models will be used elsewhere we need to converge to an acceptable compromise. One thing is clear: users no longer need to deal with arguments but variables and no longer with macros but with setups. Of course providing backward compatibility is a bit of a pain here.

The code that deals with descriptions, enumerations and notes was already done in a MkIV way, which means that they run on top of lists as storage and use the generic numbering mechanism. However, they had their own inheritance support code and moving to the generic code was a good reason to look at them again. So, now we have a new hierarchy: constructs, descriptions, enumerations and notations where notations are hooked into the (foot)note mechanisms.

These mechanisms share the rendering code but operate independently (which was the main challenge). I did explore the possibility of combining the code with lists as there are some similarities but the usual rendering is too different as in the interface (think of enumerations with optional local titles, multiple notes that get broken over pages, etc.). However, as they are also stored in lists, users can treat them as such and reuse the information when needed (which for instance is just an alternative way to deal with end notes).

At some point math formula numbering (which runs on top of enumerations) might get its own construct base. Math will be revised when we consider the time to be ripe for it anyway.

The reference mechanism is largely untouched as it was already doing well, but better support has been added for automatic cross-document referencing. For instance it is now easier to process components that make up a product and still get the right numbering and cross referencing in such an instance.

Float numbering, placement and delaying can all differ per output routine (single column, multicolumn, columnset, etc.). Some of the management has moved to Lua but most is just a job for TEX. The better some support mechanisms become, the less code we need here.

Registers will get the same treatment as lists: even more user control than is already possible. Being a simple module this is a relatively easy task, something for a hot summer day. General numbering is already fine as are block moves so they come last. The XML export and PDF tagging is also controlled from this category.

5.41 supp: support code

Support modules are similar to system ones (discussed later) but on a slightly more abstract level. There are not that many left now so these might as well become system modules at some time. The most important one is the one dealing with boxes. The biggest change there is that we use more private registers. I'm still not sure what to do with the visual debugger code. The math-related code might move to the math category.

5.42 symb: symbols

The symbol mechanisms organizes special characters in groups. With Unicode-related fonts becoming more complete we hardly need this mechanism. However, it is still the abstraction used in converters (for instance footnote symbols and interactive elements). The code has been cleaned up a bit but generally stays as is.

5.43 syst: tex system level code

Here you find all kinds of low-level helpers. Most date from early times but have been improved stepwise. We tend to remove obscure helpers (unless someone complains loudly) and add new ones every now and then. Even if we would strip down ConTEXt to a minimum size, these modules would still be there. Of course the bootstrap code is also in this category: think of allocators, predefined constants and such.

5.44 tabl: tables

The oldest table mechanism was a quite seriously patched version of TABLE and finally the decision has been made to strip, replace and clean up that bit. So, we have less code, but more features, such as colored columns and more.

The (in-stream) tabulate code is mostly unchanged but has been optimized (again) as it is often used. The multipass approach stayed but is somewhat more efficient now.

The natural table code was originally meant for XML processing but is quite popular among users. The functionality and code is frozen but benefits from optimizations in other areas. The reason for the freeze is that it is pretty complex multipass code and we don't want to break anything.

As an experiment, a variant of natural tables was made. Natural tables have a powerful inheritance model where rows and cells (first, last, ...) can be set up as a group but that is rather costly in terms of runtime. The new table variant treats each column, row and cell as an instance of `\framed` where cells can be grouped arbitrarily. And, because that is somewhat extreme, these tables are called x-tables. As much of the logic has been implemented in Lua and as these tables use buffers (for storing the main body) one could imagine that there is some penalty involved in going between TEX and Lua several times, as we have a two, three or four pass mechanism. However, this mechanism is surprisingly fast compared to natural tables. The reason for writing it was not only speed, but also the fact that in a project we had tables of 50 pages with lots of spans and such that simply didn't fit into TEX's memory any more, took ages to process, and could also confuse the float splitter.

Line tables ... well, I will look into them when needed. They are nice in a special way, as they can split vertically and horizontally, but they are seldom used. (This table mechanism was written for a project where large quantities of statistical data had to be presented.)

5.45 task: lua tasks

Currently this is mostly a place where we collect all kinds of tasks that are delegated to Lua, often hooked into callbacks. No user sees this code.

5.46 toks: token lists

This category has some helpers that are handy for tracing or manuals but no sane user will ever use them, I expect. However, at some point I will clean up this old MkIV mess. This code might end up in a module outside the core.

5.47 trac: tracing

A lot of tracing is possible in the Lua code, which can be controlled from the TEX end using generic enable and disable commands. At the macro level we do have some tracing but this will be replaced by a similar mechanism. This means that many `\tracewhatevertrue` directives will go away and be replaced. This is of course introducing some incompatibility but normally users don't use this in styles.

5.48 type: typescripts

We already mentioned that typescripts relate to fonts. Traditionally this is a layer on top of font definitions and we keep it this way. In this category there are also the definitions of typefaces: combinations of fonts. As we split the larger into smaller ones, there are many more files now. This has the added benefit that we use less memory as typescripts are loaded only once and stored permanently.

5.49 typo: typesetting and typography

This category is rather large in MkIV as we move all code into here that somehow deals with special typesetting. Here we find all kinds of interesting new code that uses Lua solutions (slower but more robust). Much has been discussed in articles as they are nice examples and often these are rather stable.

The most important new kid on the block is margin data, which has been moved into this category. The new mechanism is somewhat more powerful but the code is also quite complex and still experimental. The functionality is roughly the same as in MkII and older MkIV, but there is now more advanced inheritance, a clear separation between placement and rendering, slightly more robust stacking, local anchoring (new). It was a nice challenge but took a bit more time than other reimplementations due to all kinds of possible interference. Also, it's not always easy to simulate TEX grouping in a script language. Even if much more code is involved, it looks like the new implementation is somewhat faster. I expect to clean up this code a couple of times.

On the agenda is not only further cleanup of all modules in this category, but also more advanced control over paragraph building. There is a parbuilder written in Lua on my machine for years already which we use for experiments and in the process a more LuaTEX-ish (and efficient) way of dealing with protrusion has been explored. But for this to become effective, some of the LuaTEX backend code has to be reorganized and Hartmut wants do that first. In fact, we can then backport the new approach to the built-in builder, which is not only faster but also more efficient in terms of memory usage.

5.50 unic: Unicode vectors and helpers

As Unicode support is now native all the MkII code (mostly vectors and converters) is gone. Only a few helpers remain and even these might go away. Consider this category obsolete and replaced by the `char` category.

5.51 util: utility functions

These are Lua files that are rather stable. Think of parsers, format generation, debugging, dimension helpers, etc. Like the data category, this one is loaded quite early.

5.52 Other TEX files

Currently there are the above categories which can be recognized by filename and prefix in macro names. But there are more files involved. For instance, user extensions can go into these categories as well but they need names starting with something like `xxxx-imp-` with `xxxx` being the category.

Then there are modules that can be recognized by their prefix: `m-` (basic module), `t-` (third party module), `x-` (XML-specific module), `u-` (user module), `p-` (private module). Some modules that Wolfgang and Aditya are working on might end up in the core distribution. In a similar fashion some seldom used core code might get moved to (auto-loaded) modules.

There are currently many modules that provide tracing for mechanisms (like font and math) and these need to be normalized into a consistent interface. Often such modules show up when we work on an aspect of ConTEXt or LuaTEX and at that moment integration is not high on the agenda.

5.53 MetaPost files

A rather fundamental change in MetaPost is that it no longer has a format (mem file). Maybe at some point it will read `.gz` files, but all code is loaded at runtime.

For this reason I decided to split the files for MkII and MkIV as having version specific code in a common set no longer makes much sense. This means that already for a while we have `.mpii` and `.mpiv` files with the latter category being more efficient because we delegate some backend-related issues to ConTEXt directly. I might split up the files for MkIV a bit more so that selective loading is easier. This gives a slight performance boost when working over a network connection.

5.54 Lua files

There are some generic helper modules, with names starting with `l-`. Then there are the `mtx-*` scripts for all kinds of management tasks with the most important one being `mtx-context` for managing a TEX run.

5.55 Generic files

This leaves the bunch of generic files that provides OpenType support to packages other than ConTEXt. Much time went into moving ConTEXt-specific code out of the way and providing a better abstract interface. This means that new ConTEXt code (we provide more font magic) will be less likely to interfere and integration is easier. Of course there is a penalty for ConTEXt but it is bearable. And yes, providing generic code takes quite a lot of time so I sometimes wonder why I did it in the first place, but currently the maintenance burden is rather low. Khaled Hosny is responsible for bridging this code to LATEX.

6 What next

Here ends this summary of the current state of ConTEXt. I expect to spend the rest of the year on further cleaning up. I'm close to halfway now. What I really like is that many users upgrade as soon as there is a new beta, and as in a rewrite typos creep in, I therefore often get a fast response.

Of course it helps a lot that Wolfgang Schuster, Aditya Mahajan, and Luigi Scarso know the code so well that patches show up on the list shortly after a problem gets reported. Also, for instance Thomas Schmitz uses the latest betas in academic book production, presentations, lecture notes and more, and so provides invaluable fast feedback. And of course Mojca Miklavec keeps all of it (and us) in sync. Such a drastic cleanup could not be done without their help. So let's end this status report with ... a big thank you to all those (unnamed) patient users and contributors.

◇ Hans Hagen
`http://pragma-ade.com`

Computing the area and winding number for a Bézier curve

Bogusław Jackowski

1 Introduction

Why would we want to compute the area or winding number for a given (closed) Bézier curve? The general answer is: in computational graphics, various measures of graphical objects may prove useful.

For example, MetaPost was equipped with a very important function, missing from Metafont: `arclength`, which computes the length of a given arc. A typical problem that can be easily solved using this function is placing uniformly spaced text along a curve.

In my font application, I needed a function that calculates a distance between two curves — this feature could be used to compute an approximation of a multi-node Bézier curve by a single Bézier arc (i.e., for simplifying curves). Another operation I needed is a Boolean function telling whether a given curve is embedded in another curve.

These operations are missing from both Metafont and MetaPost, although they are feasible in Metafont due to its bitmap operations.

A useful distance between curves `a` and `b` can be computed as the number of pixels that receive a nonzero value when filling the curve `a--b--cycle` (for a given resolution). Also, checking two closed curves `a` and `b` for mutual embedding could be calculated (again, for a given resolution) by filling the curve `a` and unfilling the curve `b` — if no pixels with a positive value remain, it means that `a` is embedded in `b`.

It should be emphasized that the result depends on the resolution, which can certainly be considered a drawback.

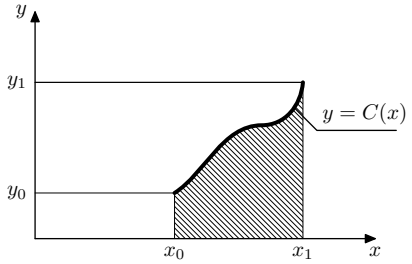
In MetaPost, bitmap operations are unavailable, hence computing the distance between curves and checking the mutual embedding of curves cannot be done simply. However, a distance between two non-intersecting curves, `a` and `b`, can be computed as the absolute value of the area surrounded by `a--b--cycle`; if the curves intersect, one has to find all intersections and compute the area for all subcycles, which is a fairly complex task (due to numerical instability).

On the other hand, a winding number — for non-touching curves — can be used for determining the mutual position of the two curves (as explained in the section below).

While bitmap operations are practically insensitive to the tangency of curves, the algorithm for computing the winding number presented here crucially is. Nevertheless, these MetaPost algorithms for computing the area and winding number may be considered counterparts of the relevant bitmap-based Metafont algorithms. Of course, I would still gladly welcome building bitmap operations into MetaPost.

2 Area enclosed by a cyclic Bézier spline

The area between the graph of a function $x \mapsto (x, C(x))$ and the x-axis is shown as the hatched region in this figure:



It can be computed as the integral

$$\int_{x_0}^{x_1} C(x)dx \tag{1}$$

If the curve is given parametrically, i.e., $t \mapsto (C^x(t), C^y(t))$, the integral (1) can be rewritten (by substituting $x = C^x(t)$, $x_0 = C^x(t_0)$, $x_1 = C^x(t_1)$, $C(x) = C(C^x(t)) = C^y(t)$, and $dx = \frac{dC^x(t)}{dt}dt$) as

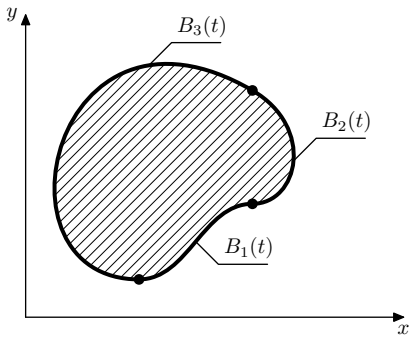
$$\int_{t_0}^{t_1} C^y(t)\,\frac{dC^x(t)}{dt}dt \tag{2}$$

Furthermore, if $t_0 \neq t_1$ and

$$(C_x(t_0), C_y(t_0)) = (C_x(t_1), C_y(t_1)),$$

i.e., the curve is cyclic, the integral (2) yields the area surrounded by the curve.

Assume that the cyclic curve is a spline composed of Bézier arcs B_1, $B_2, \ldots, B_n$ (each defined for $0 \leq t \leq 1$). The area of the region surrounded by the spline shown here:



is the sum of integrals:

$$\sum_{i=1}^{n} \int_0^1 B_i^y(t)\frac{dB_i^x(t)}{dt}\,dt$$

In the following, I'll skip the index i, as calculations are exactly the same for each i; the functions $B(t) = (B^x(t), B^y(t))$ are third-degree polynomials:

$$B(t) = b_0(1-t)^3 + 3b_1(1-t)^2\,t + 3b_2(1-t)\,t^2 + b_3t^3$$

where $b_0 = (b_0^x, b_0^y)$, $b_1 = (b_1^x, b_1^y)$, $b_2 = (b_2^x, b_2^y)$, $b_3 = (b_3^x, b_3^y)$ are points in the plane; b_0, b_3 are the nodes and b_1, b_2 are the control points of the Bézier arc B.

The computation of the antiderivative of the function $B^y(t)\frac{dB^x(t)}{dt}$ (a fifth-degree polynomial) is an elementary task (actually, it suffices to know that a derivative of t^n is nt^{n-1} and, thus, the integral of t^n is $\frac{1}{n+1}t^{n+1}$). Skipping tedious calculations, I'll present the final formula:

$$\begin{aligned}20\int_0^1 B^y(t)\frac{dB^x(t)}{dt}\,dt = {}& (b_1^x-b_0^x)(10b_0^y+6b_1^y+3b_2^y+b_3^y)\\ &+(b_2^x-b_1^x)(4b_0^y+6b_1^y+6b_2^y+4b_3^y)\\ &+(b_3^x-b_2^x)(b_0^y+3b_1^y+6b_2^y+10b_3^y)\end{aligned}$$

This formula stemmed from the discussion between Daniel Luecking and Laurent Siebenmann on the Metafont/MetaPost discussion list (`metafont@ens.fr`, 2000; presently the MetaPost discussion list is hosted by TUG — `http://lists.tug.org/metapost`). Luecking made a crucial observation that three real multiplications per Bézier arc sufficient to compute the area surrounded by a Bézier spline; division of the whole sum by 20 is a constant cost and thus can be neglected. Integer multiplication can be replaced by operations usually faster than real multiplication (e.g., $10a = 8a + 2a$, $8a = a$ shifted left by 3 bits, $2a = a$ shifted left by 1 bit).

Of course, such an optimization of the arithmetic operations makes sense only in a "production" implementation of the algorithm. The implementation at the level of Metafont/MetaPost macros can be neither efficient nor precise. Nevertheless, the following code may sometimes prove useful:

```
% p is a B\'ezier segment; result = \int y dx
vardef area(expr p) =
  save xa, xb, xc, xd, ya, yb, yc, yd;
  (xa,20ya) = point 0 of p;
  (xb,20yb) = postcontrol 0 of p;
  (xc,20yc) = precontrol 1 of p;
  (xd,20yd) = point 1 of p;
    (xb-xa)*(10ya + 6yb + 3yc +   yd)
   +(xc-xb)*( 4ya + 6yb + 6yc +  4yd)
   +(xd-xc)*(  ya + 3yb + 6yc + 10yd)
 enddef;

% P is a cyclic path; result = area of interior
vardef Area(expr P) =
 area(subpath (0,1) of P)
  for t=1 upto length(P)-1:
    + area(subpath (t,t+1) of P) endfor
enddef;
```

Observe that the macro *Area* computes a signed area: negative for counterclockwise-oriented curves, and positive for clockwise-oriented ones. As a consequence, a non-trivial curve with self-intersection(s) (e.g., eight-shaped) may surround a region with the area equal to zero.

Observe also that the calculations can be carried out with respect to the y-axis, thus the following code

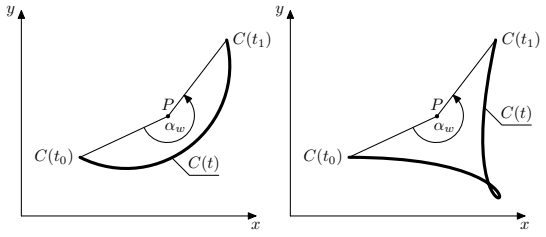
```
% p is a B\'ezier segment; result = \int y dx
vardef area(expr p) =
 save xa, xb, xc, xd, ya, yb, yc, yd;
 (-20xa,ya) = point 0 of p;
 (-20xb,yb) = postcontrol 0 of p;
 (-20xc,yc) = precontrol 1 of p;
 (-20xd,yd) = point 1 of p;
   (yb-ya)*(10xa + 6xb + 3xc +   xd)
  +(yc-yb)*( 4xa + 6xb + 6xc +  4xd)
  +(yd-yc)*(  xa + 3xb + 6xc + 10xd)
enddef;

% P is a cyclic path; result = area of interior
vardef Area(expr P) =
 area(subpath (0,1) of P)
  for t=1 upto length(P)-1:
    + area(subpath (t,t+1) of P) endfor
enddef;
```

will yield the same results as the former (within the accuracy of rounding errors).

3 A winding number for Bézier splines

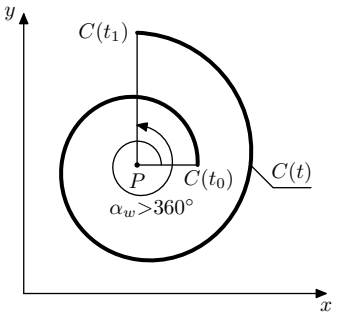
Assume that we are given a point P in the plane and the planar curve $C(t)$ defined for $t_0 \le t \le t_1$. The total angle encircled by the radius $P\,C(t)$ as t runs from t_0 to t_1 we will call the *winding angle* and denote by α_w:



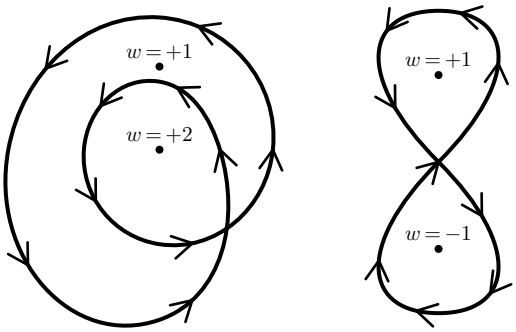
This winding angle is insensitive to certain local properties of the curve $C(t)$ (e.g., local loops): in the figures above, the *winding* angle is the same in both cases (assuming the same points P, $C(t_0)$ and $C(t_1)$).

The winding angle is positive if the point P lies to the right with respect to the point traversing the curve, and negative otherwise.

Of course, the absolute value of a winding angle can be larger than 360°:



For cyclic curves, the *winding angle* is always a multiple of 360°, i.e., $\alpha_w = 360° \, w$, where w is an integer. This entity w is called the *winding number* (for a given point and curve).



In the following, we will focus our attention on cyclic Bézier splines.

The idea of the algorithm computing the *winding number* for Bézier splines is due to Laurent Siebenmann (`metafont@ens.fr`, 2000; now at `http://lists.tug.org/metapost`). Siebenmann's solution, however, was MetaPost-oriented — it exploited heavily the operation *arctime*, available in MetaPost but unavailable in Metafont. Below, I'll present an algorithm based on the same idea but referring to more elementary properties of a Bézier segment.

For a given point P and a Bézier spline C, we will try to find the *winding angle* by measuring the *winding angles* for a discrete series of time points. First, we will try to measure the angles between nodes 0, 1, 2, ..., n of the spline C. If the relevant Bézier segments are appropriately short, the sum of the angles yields the total *winding angle*. The problem arises when the Bézier arc is too long — see, e.g., the leftmost panel of the first figure (the angle $C(t_0)\,P\,C(t_1)$ equals $360° - \alpha_w$).

The main observation of Siebenmann is as follows: if the length of the subarc $C(t)$ for $t_0 \le t \le t_1$ is shorter than the length of the longer of the radii $P\,C(t_0)$ and $P\,C(t_1)$, than we can safely assume that the (acute) angle between $P\,C(t_0)$ and $P\,C(t_1)$ is the *winding angle*. In fact, we do not need to know the exact length of the arc — an approximation suffices. If B_a, B_b, B_c, and B_d are the points defining a Bézier arc B (i.e., B_a and B_d are its nodes, B_b and B_c are its control points), then

$$|B_a\,B_b| + |B_b\,B_c| + |B_c\,B_d| \ge |B|$$

(with $|\ldots|$ denoting the length of an interval and the length of a Bézier arc). In other words, we can safely use the left-hand side of the above inequality instead of the true value of the arc length in the computation of the winding angle or winding number.

The algorithm can be expressed in pseudo-code as follows:

input: a point P and a Bézier spline B,
  consisting of segments B_1, $B_2, \ldots, B_n$
output: α_w — the winding angle for P and B
procedure *windingangle*(P, B)
  if B is a single segment
    let B_a, B_b, B_c, B_d be the consecutive
      control nodes of the segment B
    if $\min(|P\,B_a|, |P\,B_d|) <$ *assumed minimal dist.*
      exit (P almost coincides with B,
        winding angle incalculable)
    fi
    if $|B_a\,B_b| + |B_b\,B_c| + |B_c\,B_d|$
      $> \max(|P\,B_a|, |P\,B_d|)$
      return *windingangle*$(P, B(0, 1//2))$
        $+$*windingangle*$(B(1//2, 1))$
    else
      return angle α between the radii $P\,B_a$
        and $P\,B_d$ $(-90° < \alpha < 90°)$
    fi
  else
    return *windingangle*$(P, B_1) + \ldots$
      $+$*windingangle*(P, B_n)
  fi
end

A possible MetaPost/Metafont implementation:

```
% B is a B\'ezier segment
vardef mock_arclength(expr B) =
 % |mock_arclength(B)>=arclength(B)|
 length((postcontrol 0 of B)-(point 0 of B))
 + length((precontrol 1 of B)-(postcontrol 0 of B))
 + length((point 1 of B)-(precontrol 1 of B))
enddef;

% P is a point, B is a B\'ezier spline
vardef windingangle(expr P,B) =
 if length(B)=1: % single segment
  save r,v;
  r0=length(P-point 0 of B);
  r1=length(P-point 1 of B);
  if (r0<2eps) and (r1<2eps):
   % MF and MP are rather inaccurate, return 0
   errhelp "Not advisable to continue.";
   errmessage "windingangle: point almost "
```

```
    & "coincides with B\'ezier segment (dist="
    & decimal(min(r0,r1)) & ")";
  0
 else:
  % v denotes both length and angle
  v := mock_arclength(B);
  % possibly too long B\'ezier arc?
  if (v>r0) and (v>r1):
   windingangle(P, subpath (0, 1/2) of B)
   + windingangle(P, subpath (1/2, 1) of B)
  else:
   v := angle((point 1 of B)-P)  %difference
        - angle((point 0 of B)-P);
   if v >= 180: v := v-360; fi   %normalize
   if v < -180: v := v+360; fi
   v                             %return
  fi
 fi
else: % multisegment spline
 windingangle(P,subpath (0,1) of B)
  for i:=1 upto length(B)-1:
    + windingangle(P,subpath (i,i+1) of B)
  endfor
fi
enddef;
```

Although the returned angle (line marked '`%return`' above) is acute, the difference of the component angles (lines at '`%difference`') can be outside the interval $\langle -180°, 180°\rangle$; hence the normalization (lines at `%normalize`).

If the operation *windingnumber* is needed for some reason, it can be implemented trivially:

```
% P is a point, B is a B\'ezier spline
vardef windingnumber (expr P,B) =
  windingangle(P,B)/360
enddef;
```

The operations *windingangle* or, equivalently, *windingnumber* can be used, e.g., for determining the mutual position of two non-intersecting cyclic curves (whether one is embedded inside the other or not):

```
tertiarydef a inside b =
 if path a:
 % |and path b|; |a| and |b| must be cyclic and
 % must not touch each other
  begingroup
   save a_,b_;
   (a_,b_) = (windingnumber(point 0 of a,b),
              windingnumber(point 0 of b,a));
   (abs(a_ - 1) < eps) and (abs(b_) < eps)
  endgroup
 else: % |numeric a and pair b|
  begingroup
   (a>=xpart b) and (a<=ypart b)
  endgroup
 fi
enddef;
```

Postscriptum

In some cases, another definition, equivalent to the one formulated above, may be useful (this formulation, given below without a proof of equivalence, is a slightly edited excerpt from Siebenmann's message):

Assume that we are given a curve C and point P. Choose at random a line segment emanating from the point P to the point W, with W outside the bounding box of C and P. Inductively examine the intersection points Q of PQ with C. Supposing these points Q are all "nondegenerate" intersections, they are also finite in number, and a sign $+1$ or -1 is associated with each. Nondegenerate means that Q is a smooth point of c and the tangent vector T to C at Q is not parallel to PQ, and that Q is not a point where C crosses itself. The sign to use is the sign of the wedge product '$(Q-P)$ wedge T', i.e.,

$$(Q-P)\cdot(T \text{ rotated } -90)$$

The sum of the signs is the *winding number*.

It is a probabilistic theorem that degenerate intersections will rarely be met.

◇ Bogusław Jackowski
Gdańsk, Poland
b_jackowski (at) gust dot org
dot pl

Three-dimensional graphics with PGF/Ti*k*Z

Keith Wolcott

Abstract

PGF and Ti*k*Z are languages for creating graphics. These packages are predominantly two-dimensional graphics packages, so three-dimensional graphing is more challenging, but still possible. A demonstration of how to draw surfaces of revolution, satellite orbits, and intersections of spheres is given. As is typical with three-dimensional graphics, the technique is to rotate in three-space and then project to the drawing surface. The mathematics involved is discussed and sample code is provided.

1 Introduction

PGF (Portable Graphics Format) is a lower-level language and Ti*k*Z is a set of higher-level macros that use PGF. Ti*k*Z is a recursive acronym for "Ti*k*Z ist kein Zeichenprogramm" or "Ti*k*Z is not a drawing program". These languages were created by Till Tantau [1]. PGF and Ti*k*Z commands are invoked as (LA)TEX macros.

Ti*k*Z is packed with features for two-dimensional drawings of lines, circles, ellipses, paths, graphs, etc. The PGF/Ti*k*Z manual [1] has many examples to facilitate learning these languages. Andrew Mertz and William Slough [2] have written a very nice sequence of examples which is an excellent way to get started using Ti*k*Z.

The graph in figure 1 is an example of a function graphed using PGF and Ti*k*Z.

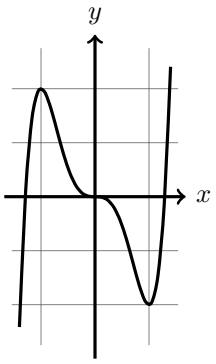


Figure 1: $f(x) = 3x^5 - 5x^3$.

The following code for this figure, and all of the examples in this paper, can be run by cutting-and-pasting into a LaTeX document of this form (be sure to use at least a 2011 version since there are compatibility issues with earlier versions of PGF/Ti*k*Z):

```
\documentclass[12pt]{article}
\usepackage{tikz}
\usepackage{ifthen}\newboolean{color}
\begin{document}
% insert code here
\end{document}
```

Here is the code to generate figure 1:

```
\begin{figure}[H]
\centering
% Set the x = a and x = b values of the domain here
% where a <= x <= b.
\def\aDomain{-1.4}
\def\bDomain{1.4}
% Set min and max values of the function
% (c <= f(x) <= d). Used for the y-axis.
\def\cRange{-2.5}
\def\dRange{2.5}
\pgfmathsetmacro\scaleAttempt{2/(\bDomain-\aDomain)}

\begin{tikzpicture}[scale= \scaleAttempt,
   domain= \aDomain : \bDomain]
\draw[very thin,color=gray]
   (1.1*\aDomain,1.1*\cRange)
   grid (1.1*\bDomain, 1.1*\dRange);
\draw[very thick, ->] (1.2*\aDomain, 0) --
   (1.2*\bDomain, 0) node[right] {$x$};
\draw[very thick, ->] (0, 1.2*\cRange) --
   (0, 1.2*\dRange) node[above] {$y$};
\draw[smooth, very thick]
   plot (\x, 3*\x^5 - 5*\x^3);
\end{tikzpicture}

\caption{$f(x) = 3x^5 - 5x^3$.}
\end{figure}
```

2 Three-dimensional graphing

The graph in figure 2 is rotated about the y-axis. The code for the figure is given in appendix A.

First we give additional examples, and then an explanation of the methods used. To create additional examples, use the eleven parameters that are

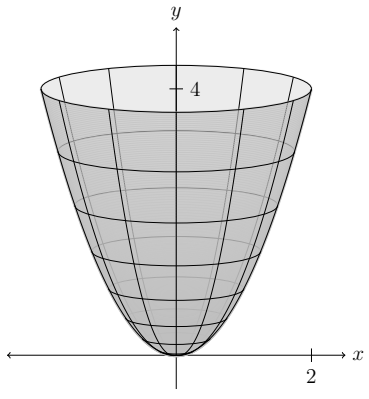


Figure 2: $f(x) = x^2$ rotated about the y-axis.

set at the beginning of the code. They are the domain (min and max), function, range (min and max), back color, front color, shading steps, `xGridSteps`, `rotationGridSteps`, and the viewing angle.

Changing the following four parameters and the figure title results in figure 3.

```
\def\bDomain{6.3}
\def\fcn{cos(\x r)} % The r means to use radians.
\def\cRange{-1}
\def\dRange{1}
```

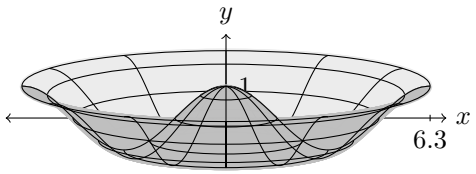


Figure 3: $f(x) = \cos x$ rotated about the y-axis.

We may not like the back and front colors so we change them both to `lightgray`. We also change the viewing angle from 10 to 15 degrees. To give some idea of how other changes can be made, we enlarge it and also comment out the x-axis and the axis number labels. The altered lines of code for figure 4 are:

```
\def\backColor{lightgray}
\def\frontColor{lightgray}

\def\phi{15} % Viewing angle of 15 degrees.

\pgfmathsetmacro\scaleAttempt{6/\bDomain}% 6 was 4.

%\draw[<->] (-\bDomain -.5, 0) -- (\bDomain + .5, 0)
%   node[right] {$x$};
%\draw (\bDomain, .1) -- (\bDomain, -.1)
%   node[below] {\bDomain};
%\pgfmathsetmacro\yLabel {cos(\phi)* \dRange}
%\draw (-.1, \yLabel) -- (.1, \yLabel)
%   node[right] {\dRange};
```

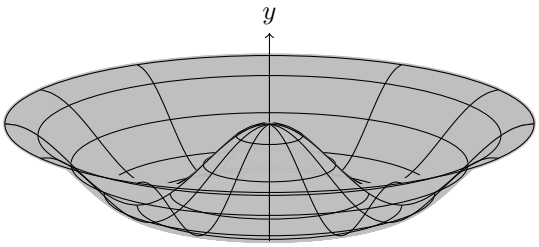


Figure 4: $f(x) = \cos x$ rotated about the y-axis.

Figure 5 is another example.

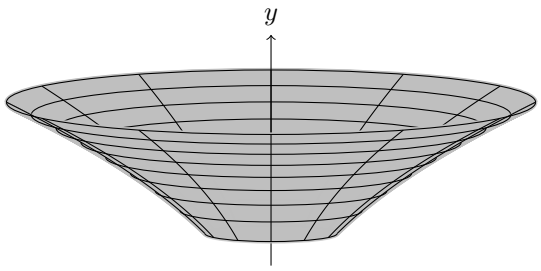


Figure 5: $f(x) = 2\sqrt{x}$, from 1 to 4, rotated about the y-axis.

Figure 6 is figure 5 rotated toward the viewer by changing the view angle from 7 to 30 degrees.

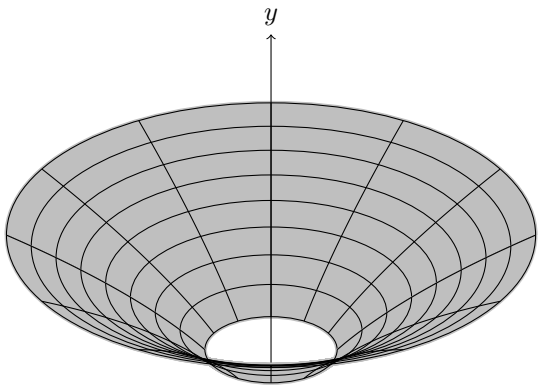


Figure 6: $f(x) = 2\sqrt{x}$, from 1 to 4, rotated about the y-axis, with a view angle of 30 degrees.

With similar code we rotate about the x-axis.

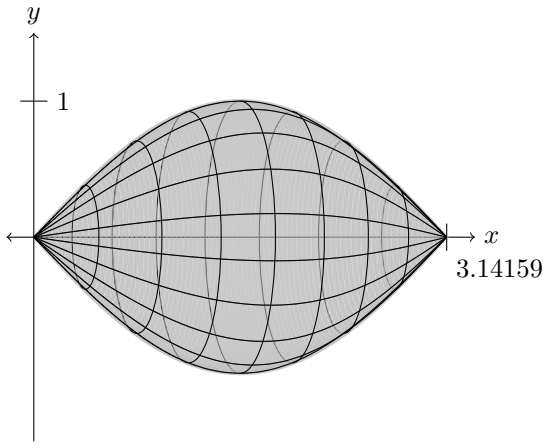


Figure 7: $f(x) = \sin x$ rotated about the x-axis.

The code for figure 7 is included in appendix B.

3 The mathematics behind the scenes

This project began with the goal of drawing two spheres and their circle of intersection. A Google search turned up Tomasz M. Trzeciak's [3] beautiful spheres (see figure 8). He very effectively draws spheres, drawing the latitude and longitude curves by creating a circle, rotating it in three-space, and then projecting to the xy-plane.

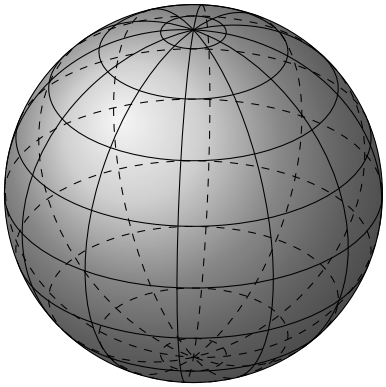


Figure 8: Beautiful sphere created by Tomasz M. Trzeciak.

We will show how to use these methods to draw the surface of revolution in figure 2.

The drawing surface is the xy-plane with the x-axis pointing to the right and the y-axis pointing up. The z-axis points toward the viewer. The rotation matrices about the x, y, and z-axes at an angle θ in the counterclockwise direction are:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix},$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix},$$

and

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

See [4] to learn about rotation matrices.

First we draw the latitude lines, which are circles with radius x drawn at height $f(x)$. To do this, draw a circle centered at the origin with radius x, but give Ti*k*Z the transformation of the plane that rotates it to give the correct viewing angle and shifts it to height $f(x)$. To find the affine transformation of the plane that does this, the following matrix rotates three-space counter-clockwise around the x-axis. Since we start with a circle in the xy-plane we need to rotate it $90 + \phi$ degrees.

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix}$$

Next, project to the xy-plane by removing the third row and third column. This gives the two-by-two matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & \cos\phi \end{pmatrix}$$

that is the transformation of the xy-plane that we tell Ti*k*Z to apply to our circle.

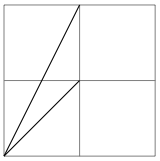
Remark: On page 253 of the PGF manual [1] version 2.10, it says that

```
\tikzset{xyplane/.estyle={cm={
   a,b,c,d,(e,f)}}}
```

uses the transformation

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}.$$

For example:



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (2,2);
\draw (0,0) -- (1,1);
\draw[cm={1,1,0,1,(0,0)}] (0,0) -- (1,1);
\end{tikzpicture}
```

The transformation should map the point (1, 1) to (2, 1) rather than (1, 2) as shown in the figure. Thus, Ti*k*Z (or the underlying PGF) is in fact using

$$(x \quad y) \begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}.$$

Since our rotation matrices operate on vectors on the left, this is the same as operating on the right with the transpose of the matrix. For a rotation matrix, the transpose of the matrix is the inverse of the matrix, which means that the direction of rotation is reversed. Thus, earlier we said that the rotation matrix $R_x(\psi)$ rotates counterclockwise, but for this application, it rotates clockwise.

In conclusion, the rotations are clockwise, and if we wish to do a sequence of rotations, the first rotation matrix is on the left.

Each circle with radius x needs to be shifted up to the correct height $f(x)$, but because of the viewing angle rotation ϕ, this gets foreshortened to

$f(x)\cos\phi$. The following code does this for 9 circles where x is each quarter unit from 0 to 2. (The ninth is an invisible point at 0.)

```
\def\fcn{\x^2}
\def\phi{-10}
\foreach \x in {0, .25, ..., 2} {
    \pgfmathsetmacro\yshift{(cos(\phi))*(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       1, 0, 0, cos(90 + \phi),(0, \yshift)}}}
    \draw[xyplane, color=black] (0, 0) circle (\x);
}
```

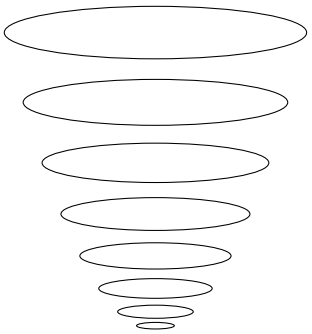


Figure 9: $f(x) = x^2$ rotated about the y-axis.

To draw the longitude lines we draw the given function $f(x)$ from 0 to 2, rotate every 30 degrees, and then change the viewing angle by ϕ degrees. Thus we use $R_y(\theta)R_x(\phi) =$

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} =$$

$$\begin{pmatrix} \cos\theta & \sin\theta\sin\phi & \sin\theta\cos\phi \\ 0 & \cos\phi & -\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{pmatrix}.$$

Next, project to the xy-plane by removing the third row and third column, resulting in the 2×2 matrix

$$\begin{pmatrix} \cos\theta & \sin\theta\sin\phi \\ 0 & \cos\phi \end{pmatrix}.$$

If θ and ϕ are both negated in the above matrix, it does not change. Thus, we will think of the rotations as being in the counterclockwise direction for positive values of θ and ϕ.

This is the transformation of the xy-plane that we tell Ti*k*Z to apply to the graph of $f(x)$.

```
\def\fcn{\x^2}
\def\phi{10}
\foreach \theta in {0, 30, ..., 360} {
 \tikzset{xyplane/.estyle={cm={
    cos(\theta),sin(\theta)*sin(\phi),
    0,cos(\phi),(0, 0)}}}
 \draw[xyplane, color=black, thin, smooth]
    plot (\x, \fcn) ;
}
```

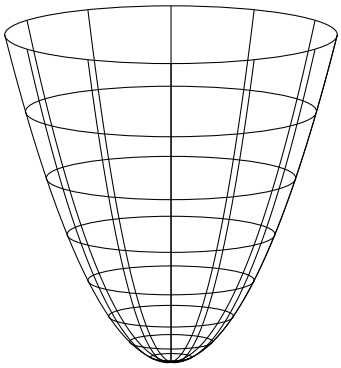


Figure 10: $f(x) = x^2$ rotated about the y-axis.

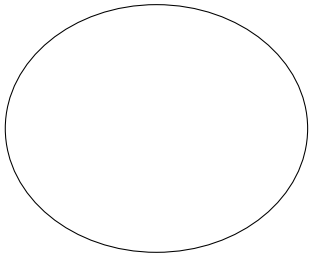
Figure 10 gives the wire frame of the surface of revolution. The remaining code for creating the final version in figure 2 is more of the same. The coloring of the front consists of the front half of many latitude curves (this slows the code down) that typically look good with opacity set to be less than 1 so that the wire frame on the back shows through a little.

4 Global Positioning System (GPS) orbits

As another example, let us draw the orbit of a GPS satellite around the earth. The orbit is inclined 55 degrees from the equator, which is the same as tilted 35 degrees from the north pole. Thus, we can start with a circle in the xy-plane and tilt it 35 degrees toward the viewer. Thus, the 35 degree tilt is a -35 degree clockwise rotation about the x-axis. The rotation matrix about the x-axis at an angle ψ in the clockwise direction is

$$R_x(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{pmatrix}.$$

Projecting to the xy-plane (removing the third row and third column), we have $\begin{pmatrix} 1 & 0 \\ 0 & \cos\psi \end{pmatrix}$. This is the transformation of the xy-plane that we tell Ti*k*Z to apply to our circle as follows. The angle ψ should be -35 degrees since we are tilting the top of the circle toward the viewer.



```
\begin{tikzpicture}
 \tikzset{xyplane/.estyle={cm={
    1,0,0,cos(-35), (0, 0)}}}
 \draw[xyplane, color=black, thin]
    (0, 0) circle (2);
\end{tikzpicture}
```

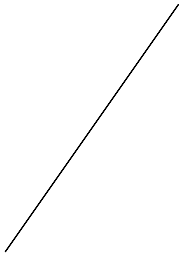
Now we wish to rotate our tilted orbit θ degrees around the y-axis. To rotate first ψ degrees about the x-axis and then θ degrees about the y-axis, we compute

$$R_x(\psi)R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ \sin\psi\sin\theta & \cos\psi & -\sin\psi\cos\theta \\ -\cos\psi\sin\theta & \sin\psi & \cos\psi\cos\theta \end{pmatrix}.$$

Next, project to the xy-plane, which is the transformation

$$\begin{pmatrix} \cos\theta & 0 \\ \sin\psi\sin\theta & \cos\psi \end{pmatrix}.$$

If θ, ψ, and ϕ are all negated in the above matrix, it does not change. Thus, we will think of the rotations as being in the counterclockwise direction for positive values of θ, ψ, and ϕ.



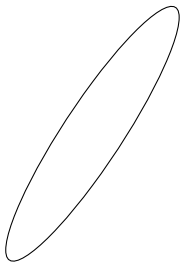
```
\begin{tikzpicture}
 \tikzset{xyplane/.estyle={cm={
    cos(90),0,sin(35)*sin(90),cos(35),(0, 0)}}}
 \draw[xyplane] (0, 0) circle (2);
\end{tikzpicture}
```

Now suppose that we wish to view this from above and thus rotate some angle ϕ around the x-axis to tilt the top of the orbit toward the viewer. This is the product $R_x(\psi)R_y(\theta)R_x(\phi)$ that has the projected matrix

$$\begin{pmatrix} \cos\theta & \sin\theta\sin\phi \\ \sin\psi\sin\theta & \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi \end{pmatrix}.$$

Again, if θ, ψ, and ϕ are all negated in the above matrix, it does not change. Thus, we will think of the rotations as being in the counterclockwise direction for positive values of θ, ψ, and ϕ.

With $\psi = 35$, $\theta = 90$, and $\phi = 20$, we rotate the top of the above circle (that looks like a line) toward the viewer 20 degrees so it looks like an ellipse again.



```
\begin{tikzpicture}
  \tikzset{xyplane/.estyle={cm={
    cos(90),sin(90)*sin(20),sin(35)*sin(90),
    cos(35)*cos(20)-sin(35)*cos(90)*sin(20),(0,0)}}}
  \draw[xyplane] (0, 0) circle (2);
\end{tikzpicture}
```

As an example of the above rotation and projection matrix, we draw the U.S. GPS (Global Positioning System) satellite orbits. There are six orbits, each tilted $\psi = 35$ degrees from the north pole and then rotated to be positioned every 60 degrees (θ) around the earth. These orbits can then be viewed from different angles ϕ. Figure 11 shows a view from the equator. The code is included in appendix C.

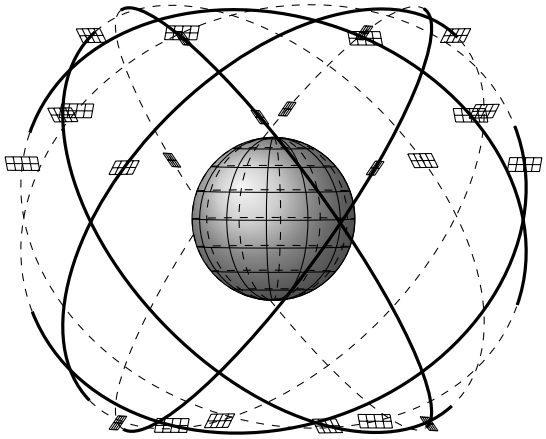


Figure 11: The U.S. GPS system. This view, looking directly at the equator, shows that the orbits never pass over the north or south poles. Each orbit has four satellites spaced 30 degrees, 105 degrees, 120 degrees, and 105 degrees apart.

By changing the viewing angle `\angEl` $= \psi$, we can get additional views. See figures 12 and 13 for two such views.

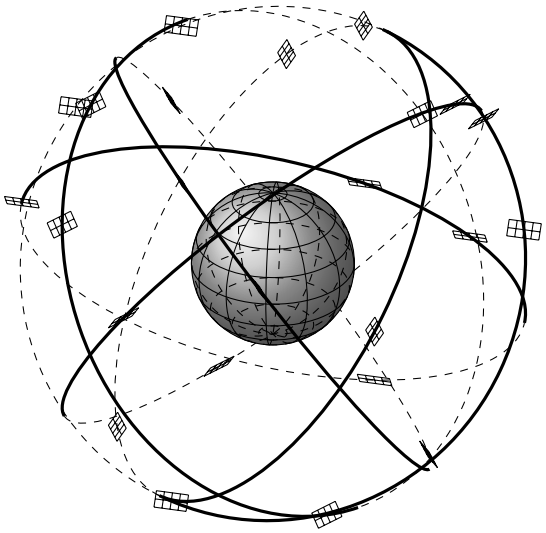


Figure 12: Looking down on the earth, 30 degrees above the equator. Dotted lines are on the back side of the orbit.

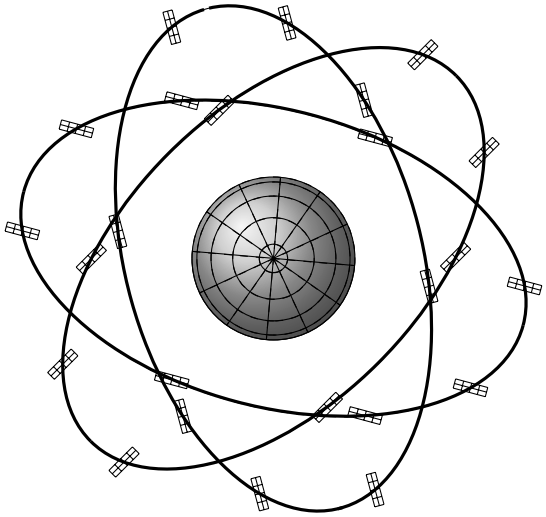


Figure 13: Looking directly down on the north pole. Pairs of orbits overlap in this view, but the front and back parts of these orbits are on opposite sides of the earth.

For comparison, figure 14 is the configuration of satellites that the European Union is using for the GPS system that they are currently implementing that has just three orbits with nine satellites in each.

5 Intersections of spheres

Rotations around an axis other than the coordinate axes can also be useful. Consider figure 15.

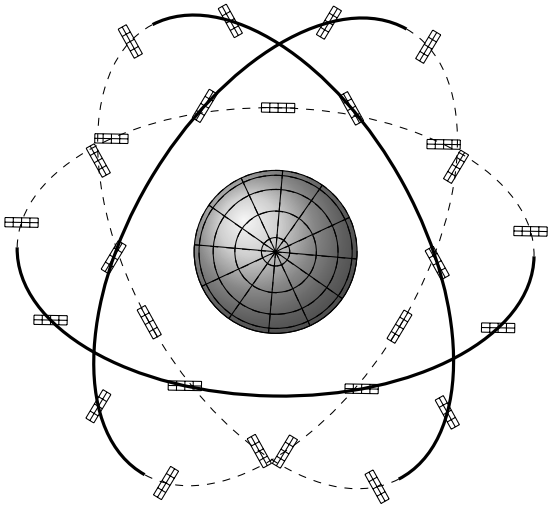


Figure 14: The European Union's planned GPS has three orbital planes, 120 degrees apart, inclined at 56 degrees, that divide the earth's surface into eight congruent spherical triangles. Each of the three orbits has nine satellites, equally spaced, 40 degrees apart.

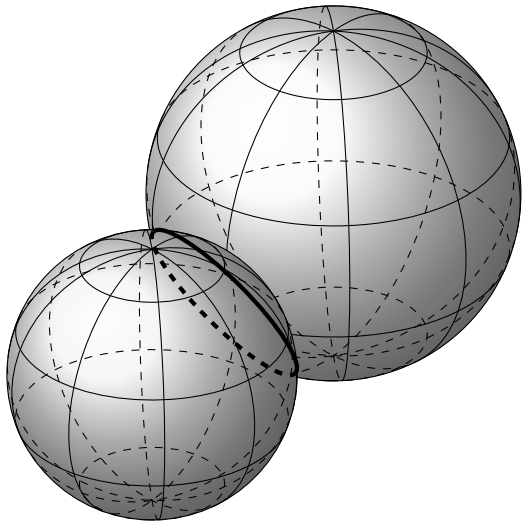


Figure 15: These two spheres intersect in the bold circle.

The goal was to draw the intersection of the two spheres. The plane of the circle of intersection is perpendicular to the vector between the two sphere centers. Thus to find the circle of intersection we rotate three-space so that the z-axis (which is perpendicular to the xy-plane) rotates to be parallel to the vector between the sphere centers. Thus the rotation axis is perpendicular to both of these vectors and we compute it by taking the cross product of them. Then we use the rotation matrix about this vector

(see [4]), project to the xy-plane, and then shift to the correct position. This figure needs more work, since it does not show the overlapping parts of the spheres correctly, but it is still an example of how to find and draw the intersection using these techniques. The code for figure 15 is included in appendix E.

6 Conclusion

The PGF and Ti*k*Z languages are predominantly for two-dimensional graphics, but with an understanding of a few rotation matrices, some three-dimensional graphics can be drawn fairly easily.

Appendix A Rotation about the y-axis

Code for figure 2. The function $f(x) = x^2$ is rotated about the y-axis.

```
\begin{figure}[h]
\begin{center}
%%%%%%%   Set function values  %%%%%%%
% Set the x = a and x = b values of the
% domain here where a <= x <= b.
\def\aDomain{0}
\def\bDomain{2}
% Set the function.
% The variable must be \x, e.g. \x^2.
\def\fcn{\x^2}
% Set min and max values of the function
% (c <= f(x) <= d). Used for the y-axis.
\def\cRange{0}
\def\dRange{4}
% Set the color of the back half.
% This can look good as a different color
% if it looks like the inside.
\def\backColor{brown}
% Set the color of the front half.  lightgray looks
% good for both back and front.
\def\frontColor{red}
% Set the number of shading circles to draw.
% More gives a more even color.
% Enter 1 for no shading.
\def\xShadingSteps{300}
% Set the number of x radius grid circles.
\def\xGridSteps{8}
% Set the number of radial grid lines.
\def\rotationGridSteps{12}
% Set the viewing elevation angle,
% which is the angle up from horizontal.
\def\phi{10}
%%%%%%%%%%%%%%%%%%%%%%%%
\pgfmathsetmacro\scaleAttempt{4/\bDomain}
\begin{tikzpicture}[scale= \scaleAttempt,
   domain= \aDomain: \bDomain]
\pgfmathsetmacro\intervalLength{\bDomain - \aDomain}
\pgfmathsetmacro\xGridStepsize{
   \intervalLength/\xGridSteps}
\pgfmathsetmacro\xShadingStepsize{
   \intervalLength/\xShadingSteps}
\pgfmathsetmacro\rotationGridStepsize{
   360/\rotationGridSteps}
% Draw the shading of the back half.
% Top half of a circle, rotated back (around x-axis)
% 90 - \phi degrees and shifted up or down
% to the correct height.
\pgfmathsetmacro\nextShadingStep{
   \aDomain + \xShadingStepsize}
\foreach \x in
   {\aDomain, \nextShadingStep, ..., \bDomain} {
    \pgfmathsetmacro\ysh {(cos(\phi))*(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       1,0,0,cos(90-\phi), (0, \ysh)}}}
    \draw[xyplane,\backColor,ultra thick,opacity=1]
       (\x, 0) arc (0:180:\x);
}
% Back longitude lines.
% Rotates graph around y-axis, then
% projects to xy-plane.
\foreach \theta in
   {0, \rotationGridStepsize, ..., 180} {
 \tikzset{xyplane/.estyle={cm={
    cos(-\theta), sin(-\theta)*sin(-\phi),
    0, cos(-\phi),  (0, 0)}}}
 \draw[xyplane, smooth] plot (\x, \fcn);
}
% Back latitude lines.
% Top half of a circle, rotated back
% (around x-axis) 90 - \phi degrees and
% shifted up or down to the correct height.
\pgfmathsetmacro\nextStep{\aDomain + \xGridStepsize}
\foreach \x in {\aDomain,\nextStep, ...,\bDomain} {
    \pgfmathsetmacro\ysh {(cos(\phi))*(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       1,0,0,cos(90-\phi), (0, \ysh)}}}
    \draw[xyplane] (\x, 0) arc (0: 180:\x);
}
% Draw the axis.
\pgfmathsetmacro\yHeight{
   \dRange + \bDomain*sin(\phi) + .5}
\draw[->] (0, \cRange - .5) -- (0, \yHeight)
   node[above] {$y$};
% Comment out the next four commands
% if you don't want an x-axis, and labels.
\draw[<->] (-\bDomain -.5, 0) -- (\bDomain + .5, 0)
   node[right] {$x$};
\draw (\bDomain, .1) -- (\bDomain, -.1)
   node[below] {\bDomain};
\pgfmathsetmacro\yLabel {cos(\phi)* \dRange}
\draw (-.1, \yLabel) -- (.1, \yLabel)
   node[right] {\dRange};

% Draw the shading of the front half.
% Top half of a circle, rotated back (around x-axis)
% 90 - \phi degrees and shifted up or down
% to the correct height.
\foreach \x in
   {\aDomain, \nextShadingStep, ..., \bDomain} {
    \pgfmathsetmacro\ysh {(cos(\phi))*(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       1,0,0,cos(90-\phi), (0, \ysh)}}}
    \draw[xyplane,\frontColor,ultra thick,
       opacity=.6]
       (-\x, 0) arc (-180:0:\x);
}
% Front longitude lines.
\foreach \theta in
   {0, \rotationGridStepsize, ..., 180} {
 \tikzset{xyplane/.estyle={cm={
    cos(\theta), sin(\theta)*sin(-\phi),
    0, cos(-\phi),  (0, 0)}}}
 \draw[xyplane,smooth] plot (\x, \fcn);
}
% Front latitude lines.
% Bottom half of a circle, rotated back
```

```
% (around x-axis) 90 - \phi degrees and
% shifted up or down to the correct height.
\foreach \x in {\aDomain, \nextStep, ..., \bDomain}{
    \pgfmathsetmacro\ysh {(cos(\phi))*(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       1,0,0,cos(90-\phi),(0, \ysh)}}}
    \draw[xyplane] (-\x, 0) arc (-180: 0:\x);
}
\end{tikzpicture}
\caption{Rotate $f(x) = x^2$ about the $y$-axis.}
\end{center}
\end{figure}
```

Remark: Since the back and front shading is drawn by drawing circles that are rotated in three-space, and the above code uses 300 such circles, it is slow (about 3 seconds). Thus, when working with the document, it is useful to set `\xShadingSteps` to 1 and then change it to 300 for the final version.

Appendix B Rotation about the x-axis

Code for figure 7. The function $\sin x$ is rotated about the x-axis.

```
\begin{figure}[h]
\begin{center}
%%%%%%%   Set function values  %%%%%%%
% Set the x = a and x = b values of the
% domain here where a <= x <= b.
\def\aDomain{0}
\def\bDomain{3.14159}
% Set the function.
% The variable must be \x, e.g. \x^2.
\def\fcn{sin(\x r)}
%\def\fcn{sqrt(\x)}
% Set min and max values of the function
% (c <= f(x) <= d). Used for the y-axis.
\def\cRange{0}
\def\dRange{1}
% Set the color of the back half.
% This can look good as a different color
% if it looks like the inside.
\def\backColor{red!70!black}
% Set the color of the front half.  lightgray looks
% good for both back and front.
\def\frontColor{red!70!black}
% Set the number of shading circles to draw.
% More gives a more even color.  Enter 1 for
% no shading; a large number makes it slow.
% Use the following two lines while editing and then
% change the speed to 100 for the final version.
%\def\speed{1}
%\pgfmathsetmacro\xShadingSteps{3* \speed}
\pgfmathsetmacro\xShadingSteps{300}
% Set the number of x radius grid circles.
\def\xGridSteps{8}
% Set the number of radial grid lines.
\def\rotationGridSteps{18}
% Set the viewing elevation angle,
% which is the angle up from horizontal.
\def\phi{15}
%%%%%%%%%%%%%%%%%%%%%%%%
\pgfmathsetmacro\scaleAttempt{3.4/\dRange}
\begin{tikzpicture}[scale= \scaleAttempt,
   domain= \aDomain: \bDomain]
\pgfmathsetmacro\intervalLength{\bDomain - \aDomain}
\pgfmathsetmacro\xGridStepsize{
   \intervalLength/\xGridSteps}
\pgfmathsetmacro\xShadingStepsize{
   \intervalLength/\xShadingSteps}
\pgfmathsetmacro\rotationGridStepsize{
   360/\rotationGridSteps}
% Draw the shading of the back half.
% Left half of a circle, rotated right
% (around y-axis) 90 - \phi degrees and
% shifted right or left to the correct height.
\pgfmathsetmacro\nextShadingStep{
   \aDomain + \xShadingStepsize}
\foreach \x in
  {\aDomain, \nextShadingStep, ..., \bDomain} {
    \pgfmathsetmacro\xsh{(cos(\phi))*({\x})}
    \pgfmathsetmacro\rad{(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       cos(\phi - 90), 0,0,1, (\xsh, 0)}}}
    \draw[xyplane,\backColor,ultra thick,opacity=.6]
       (0, \rad) arc (90 : 270 : \rad);
}
% Back longitude lines.
% Rotates graph around y-axis,
% then projects to xy-plane.
\foreach \theta in
   {0, \rotationGridStepsize, ..., 180} {
 \tikzset{xyplane/.estyle={cm={
    cos(\phi), 0, sin(\theta)*sin(\phi),
    cos(\theta),  (0, 0)}}}
 \draw[xyplane,smooth] plot (\x, \fcn) ;
}
% Back latitude lines.
% Left half of a circle, rotated right
% (around y-axis) 90 - \phi degrees and
% shifted right or left to the correct height.
\pgfmathsetmacro\nextStep{\aDomain + \xGridStepsize}
\foreach \x in {\aDomain,\nextStep, ...,\bDomain} {
    \pgfmathsetmacro\xsh{(cos(\phi))*({\x})}
    \pgfmathsetmacro\rad{(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       cos(\phi - 90), 0,0,1,(\xsh, 0)}}}
    \draw[xyplane,black,thin,opacity=1]
       (0, \rad) arc (90 : 270 : \rad);
}
% Draw the axis.
\pgfmathsetmacro\xdim{
   \bDomain + \dRange*sin(\phi) + .5}
\draw[->] (0, -\dRange - .5) -- (0, \dRange + .5)
   node[above] {$y$};
% Comment out the next four commands
% if you don't want an x-axis, and labels.
\draw[<->] (\aDomain -.5, 0) -- (\xdim, 0)
   node[right] {$x$};
\pgfmathsetmacro\xLabel{cos(\phi)*\bDomain}
\draw (\xLabel, .1) -- (\xLabel, -.1)
   node[below right] {\bDomain};
\draw (-.1, \dRange) -- (.1, \dRange)
   node[right] {\dRange};

% Draw the shading of the front half.
% Right half of a circle, rotated right
% (around y-axis) 90 - \phi degrees and
% shifted right or left to the correct height.
\foreach \x in
  {\aDomain, \nextShadingStep, ..., \bDomain} {
    \pgfmathsetmacro\xsh{(cos(\phi))*({\x})}
    \pgfmathsetmacro\rad{(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       cos(\phi - 90),0,0,1,(\xsh, 0)}}}
```

```
   \draw[xyplane,\frontColor,ultra thick,opacity=.6]
       (0, -\rad) arc (-90 : 90 : \rad);
}
% Front longitude lines.
\foreach \theta in
   {0, \rotationGridStepsize, ..., 180} {
 \tikzset{xyplane/.estyle={cm={
    cos(\phi), 0,
    sin(\theta)*sin(\phi),cos(\theta),(0, 0)}}}
 \draw[xyplane,smooth] plot (\x, \fcn) ;
}
% Front latitude lines.
% Right half of a circle, rotated right
% (around y-axis) 90 - \phi degrees and
% shifted right or left to the correct height.
\foreach \x in {\aDomain, \nextStep, ..., \bDomain}{
    \pgfmathsetmacro\xsh{(cos(\phi))*({\x})}
    \pgfmathsetmacro\rad{(\fcn)}
    \tikzset{xyplane/.estyle={cm={
       cos(\phi-90),0,0,1, (\xsh, 0)}}}
    \draw[xyplane] (0, -\rad) arc (-90 : 90 : \rad);
}
\end{tikzpicture}
\caption{$f(x)=\sin{x}$ rotated about the $x$-axis.}
\label{rot1x}
\end{center}
\end{figure}
```

Appendix C GPS satellites

Code for figure 11, the U.S. GPS satellite orbits.

```
% GPS satellite orbits.
\begin{tikzpicture}[scale=.77]
\def\R{1.4} % sphere radius
\def\orbitRadius{3.172*\R}
\def\angEl{1} % elevation angle
\def\x{0} % x coordinate of center
\def\y{0} % y coordinate of center
\def\z{0} % z coordinate of center
% First tilt the orbit from the north
% pole (rotate about the x-axis).
\pgfmathsetmacro\psi{35}
% Second, rotate around the y-axis.
\pgfmathsetmacro\firstTheta{-135}
% Third, rotate about the x-axis.
\pgfmathsetmacro\phi{\angEl}
\draw[color=red, fill=blue, opacity=.15]
   (0, 0) circle (\orbitRadius);
% Set the variables, theta, c = color, angVis,
% and \thetaSatShift for each of 6 orbits.
\foreach \theta/\c in{\firstTheta/red,
   \firstTheta+60/blue,
   \firstTheta+2*60/green,
   \firstTheta+3*60/black,
   \firstTheta+4*60/cyan,
   \firstTheta+5*60/brown}{
% Set the drawing plane affine transformation.
\tikzset{xyplane/.estyle={cm={
   cos(\theta),sin(\theta)*sin(\phi),
   sin(\theta)*sin(\psi),cos(\psi)*cos(\phi)-
   sin(\psi)*cos(\theta)*sin(\phi),(0, 0)}}}
    % Draw the back half of the orbit.
    \getFrontArcStartPosition\angle\anglex{
       \psi}{\theta}{\phi}
   \pgfmathtruncatemacro\angleInt{\angle}
    \ifthenelse{\angleInt < -180}
    {\pgfmathsetmacro\angleInt{\angleInt + 360}}
    {}
    \draw[xyplane, dashed, color=\c]
       (\angleInt -180: \orbitRadius)
       arc (\angleInt -180: \angle: \orbitRadius);
}
% Draw the earth.
\tikzset{current plane/.estyle={cm={1,0,0,1,(0,0)}}}
\filldraw[current plane][shift={(\x, \y)}]
   [ball color=blue,opacity=.7] (0,0,0) circle (\R);
\foreach \t in {-80,-60,...,80} {
   \DrawLatitudeCircle[\R]{\t}{\x}{\y}}
\foreach \t in {-5,-35,...,-175} {
   \DrawLongitudeCircle[\R]{\t}{\x}{\y}}
\pgfmathsetmacro\orbitBaseAngle{30}
% Draw the front half of the orbit.
\foreach \theta/\c in {\firstTheta/red,
   \firstTheta+60/blue,
   \firstTheta+2*60/green,
   \firstTheta+3*60/black,
   \firstTheta+4*60/cyan,
   \firstTheta+5*60/brown}{
% Set the drawing plane affine transformation again.
   \tikzset{xyplane/.estyle={cm={cos(\theta),
      sin(\theta)*sin(\phi),sin(\theta)*sin(\psi),
      cos(\psi)*cos(\phi)-sin(\psi)*cos(\theta)*
      sin(\phi),(0,0)}}}
% Draw the front half of the orbit.
\getFrontArcStartPosition\angle\anglex{
   \psi}{\theta}{\phi}
\pgfmathtruncatemacro\angleInt{\angle}
\ifthenelse{\angleInt > 180}
     {\pgfmathsetmacro\angleInt{\angleInt - 360}}
     {}
\draw[xyplane,very thick,color=\c]
   (\angleInt:\orbitRadius) arc
   (\angleInt:\angleInt+180:\orbitRadius);
     % Draw the satellites.
     \foreach \thetaSat in {\orbitBaseAngle,
   \orbitBaseAngle + 30, \orbitBaseAngle + 135,
   \orbitBaseAngle + 255} {
          \pgfmathsetmacro\xsh{
           (7/1)*\orbitRadius*cos(\thetaSat)}
           \pgfmathsetmacro\ysh{
    (7/1)*\orbitRadius*sin(\thetaSat)}
          \draw[xyplane,color=\c,scale=1/7][shift=
             {(\xsh,\ysh)}](-2,-1) grid (2,1);
    }
}
\end{tikzpicture}
```

This code requires some helper functions written by Tomasz M. Trzeciak [3] for drawing spheres. For completeness, these functions are listed below. Place them just before `\begin{document}`.

```
\newcommand\pgfmathsinandcos[3]{%
  \pgfmathsetmacro#1{sin(#3)}%
  \pgfmathsetmacro#2{cos(#3)}%
}
\newcommand\LongitudePlane[3][current plane]{%
  \pgfmathsinandcos\sinEl\cosEl{#2} % elevation
  \pgfmathsinandcos\sint\cost{#3} % azimuth
  \tikzset{#1/.estyle={cm={
     \cost,\sint*\sinEl,0,\cosEl,(0,0)}}}
}
\newcommand\LatitudePlane[3][current plane]{%
  \pgfmathsinandcos\sinEl\cosEl{#2} % elevation
  \pgfmathsinandcos\sint\cost{#3} % latitude
  \pgfmathsetmacro\yshift{\cosEl*\sint}
```

```
  \tikzset{#1/.estyle={cm={
     \cost,0,0,\cost*\sinEl,(0,\yshift)}}} %
}
\newcommand\DrawLongitudeCircle[4][1]{
 \LongitudePlane{\angEl}{#2}
 \tikzset{current plane/.prefix style={scale=#1}}
  % angle of "visibility"
 \pgfmathsetmacro\angVis{
    atan(sin(#2)*cos(\angEl)/sin(\angEl))} %
 \draw[shift={(#3, #4)}][current plane]
    (\angVis:1) arc (\angVis:\angVis+180:1);
 \draw[shift={(#3, #4)}][current plane,dashed]
    (\angVis-180:1)arc(\angVis-180:\angVis:1);
}
\newcommand\DrawLatitudeCircle[4][1]{
 \LatitudePlane{\angEl}{#2}
 \tikzset{current plane/.prefix style={scale=#1}}
 \pgfmathsetmacro\sinVis{
    sin(#2)/cos(#2)*sin(\angEl)/cos(\angEl)}
 % angle of "visibility"
 \pgfmathsetmacro\angVis{
    asin(min(1,max(\sinVis,-1)))}
 \draw[shift={(#3, #4)}][current plane]
    (\angVis:1) arc (\angVis:-\angVis-180:1);
 \draw[shift={(#3, #4)}][current plane,dashed]
    (180-\angVis:1)arc(180-\angVis:\angVis:1);
}
```

This uses a macro `\getFrontArcStartPosition` to compute which parts of the orbit arcs are on the front side of the orbit so they can be drawn last and the back side can be drawn first with dotted lines. There is likely an easier way to do this, but this solution involves using the spherical law of sines on various triangles on the sphere. For completeness, an explanation of the formulas follows the code.

```
\newcommand\getFrontArcStartPosition[5]{
% Theta must be between -180 and 180.
\pgfmathtruncatemacro\psiInt{#3}
\pgfmathtruncatemacro\thetaInt{#4}
\pgfmathtruncatemacro\phiInt{#5}
\pgfmathtruncatemacro\psiTemp{\psiInt}
\ifthenelse{\thetaInt < 0}
% Negate theta and negate the results at the end.
    {\pgfmathtruncatemacro\thetaTemp{-\thetaInt}}
    {\pgfmathtruncatemacro\thetaTemp{\thetaInt}}
\pgfmathtruncatemacro\phiTemp{\phiInt}
\pgfmathsetmacro\anglexTemp{atan(sin(\thetaTemp)/
   (cos(\thetaTemp)*sin(\psiTemp)))}

\ifthenelse{\thetaTemp > 90}
    {\pgfmathsetmacro\anglexTemp{\anglexTemp + 180}}
    {}
\pgfmathsetmacro\result{atan(sin(\thetaTemp)*
   cos(\phiTemp)*sin(\anglexTemp)/
(sin(\anglexTemp)*cos(\psiTemp)*sin(\phiTemp)
+cos(\anglexTemp)*cos(\phiTemp)*sin(\thetaTemp)))}
\pgfmathsetmacro\specialAngle{(cos(\phiTemp)*
   cos(\phiTemp)-cos(\psiTemp)*cos(\psiTemp))/
   (cos(\phiTemp)*cos(\phiTemp)*
   sin(\psiTemp)*sin(\psiTemp))}
\ifthenelse{\phiInt < \psiInt}{
    \pgfmathparse{sqrt(\specialAngle)}
    \pgfmathsetmacro\specialAngle{
       asin(-\pgfmathresult)+180}
    \ifthenelse{\thetaTemp > \specialAngle}{
        \pgfmathsetmacro\result{\result + 180}
        \pgfmathsetmacro#1{\result}}
    {
        \pgfmathsetmacro#1{\result}}}
{}
% Negate the results if theta is negative.
\ifthenelse{\thetaInt < 0}{
    \pgfmathsetmacro#1{-\result}
    \pgfmathsetmacro#2{-\anglexTemp}}
{
    \pgfmathsetmacro#1{\result}
    \pgfmathsetmacro#2{\anglexTemp}}
}% End of \getFrontArcStartPosition function.
```

Appendix D Explanation of formulas in `\getFrontArcStartPosition`

The function `\getFrontArcStartPosition` in appendix C is used to find which parts of an orbit are on the front and which on the back, so that we can draw the back as a dotted line.

Figure 16 shows a red orbit that is tilted ψ degrees from the north pole and then rotated θ degrees with the viewing angle tilted ϕ degrees as the orbits are for figure 11. Our goal is to find the length of arc $DC = x'$ since that is where the orbit comes around to the front of the sphere.

The spherical law of sines says that for a triangle on a sphere

$$\frac{\sin A}{\sin a} = \frac{\sin B}{\sin b} = \frac{\sin C}{\sin c}$$

where a, b, and c are the three angles of the triangle and A, B, and C are the three corresponding opposite side lengths (which are measured as an angle from the center of the sphere).

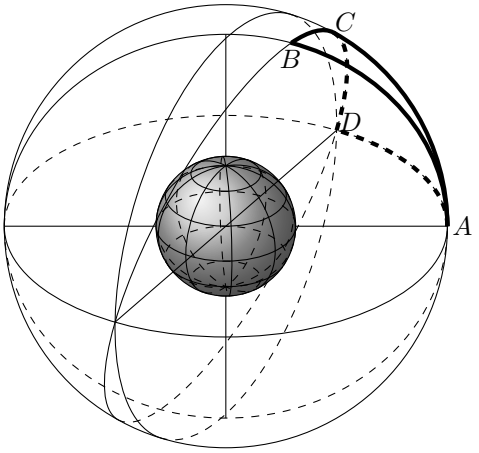


Figure 16: Given the orbit containing points B, C, and D, find the angular distance from D to C. If this distance is found, then the orbit is drawn with a solid arc from that point at C for 180 degrees and then for another 180 degrees as a dashed line.

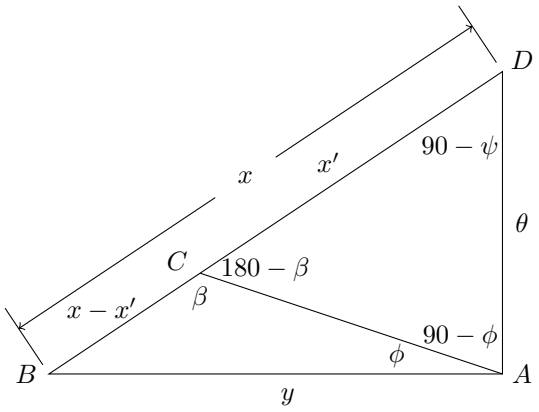


Figure 17: Triangles of figure 16 drawn in the plane.

Using the law of sines on the large triangle ABD, we have that

$$\frac{\sin y}{\sin(90-\psi)} = \frac{\sin x}{\sin 90}. \tag{1}$$

Using the law of sines on the lower small triangle ABC, we have that

$$\frac{\sin\beta}{\sin y} = \frac{\sin\phi}{\sin(x-x')}. \tag{2}$$

Simplifying and solving (1) for $\sin y$ and substituting into (2) and solving for $\sin\beta$, results in

$$\sin\beta = \frac{\cos\psi\sin x\sin\phi}{\sin(x-x')}. \tag{3}$$

Using the law of sines on the upper small triangle ACD, we have that

$$\frac{\sin(180-\beta)}{\sin\theta} = \frac{\sin(90-\phi)}{\sin x'}. \tag{4}$$

Solving (4) for $\sin\beta$ and setting equal to the right side of (3) results in

$$\frac{\sin\theta\cos\phi}{\sin x'} = \frac{\cos\psi\sin x\sin\phi}{\sin(x-x')}$$

or

$$\frac{\sin\theta\cos\phi}{\sin x'} = \frac{\cos\psi\sin x\sin\phi}{\sin x\cos x' - \cos x\sin x'}.$$

Cross multiplying, dividing by $\cos x'$ and solving for $\tan x'$ gives

$$\tan x' = \frac{\sin\theta\cos\phi\sin x}{\sin\theta\cos\phi\cos x + \cos\psi\sin\phi\sin x}.$$

Thus we can compute x' in terms of x. In order to compute x, see figure 18 where we have added points N and E.

Point E is chosen such that angle NEB is a right angle. Triangle NED is drawn in figure 19.

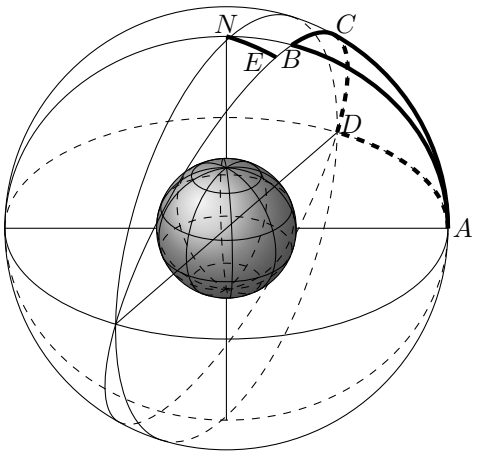


Figure 18: This is the same as figure 16 with added points N and E where segment NE is perpendicular to both red orbits. Triangle NED is used to find the value of x which is the arc BD.

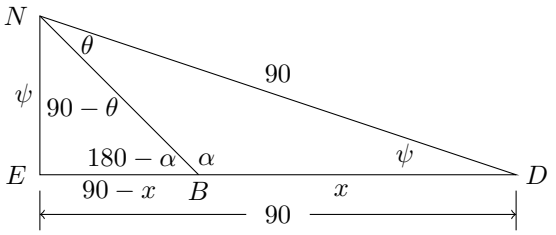


Figure 19: Triangle NED of figure 18 drawn in the plane.

The law of sines applied to triangle NBD on the right gives

$$\sin\alpha = \frac{\sin\theta}{\sin x}. \tag{5}$$

The law of sines applied to triangle NEB on the left gives $\sin(180-\alpha) =$

$$\sin\alpha = \frac{\sin\psi\sin(90-\theta)}{\sin(90-x)} = \frac{\sin\psi\cos\theta}{\cos x}. \tag{6}$$

Setting the values of $\sin\alpha$ equal from (5) and (6) and solving for $\tan x$ gives

$$\tan x = \frac{\tan\theta}{\sin\psi}.$$

Thus we now have x in terms of θ and ψ. This explains the formulas used for $0 \le \theta \le 90$ in the macro `\getFrontArcStartPosition`. In the case that $\psi \le \phi$, the same formulas work for when $90 \le \theta \le 180$. There are some complications when $\psi > \phi$ and $90 \le \theta \le 180$. In this case, when x' exceeds 90 degrees there is a sign change in the computation. Some more spherical trigonometry reveals that this

happens when

$$\sin^2\theta = \frac{\cos^2\phi - \cos^2\psi}{\cos^2\phi \sin^2\psi}$$

which we use in `\getFrontArcStartPosition` to compute the value of θ where this change occurs. This allows us to compute x' for all $0 \le \theta \le 180$. For $-180 \le \theta \le 0$, we use symmetry and return the negative of the x' computed for $|\theta|$.

Appendix E Intersection of two spheres

Code for figure 15.

```
% The intersection of two spheres.
\begin{tikzpicture}[scale=.8]
% Draw the first sphere.
\def\Rb{3.1} % sphere radius
\def\angEl{30} % elevation angle
\def\xb{3} % x coordinate of center
\def\yb{3} % y coordinate of center
\def\zb{-1} % z coordinate of center
\filldraw[shift={(\xb, \yb)}][ball color= blue]
   (0, 0, 0) circle (\Rb);
\foreach \t in {-60,-20,...,80} {
   \DrawLatitudeCircle[\Rb]{\t}{\xb}{\yb}}
\foreach \t in {-5,-45,...,-175} {
   \DrawLongitudeCircle[\Rb]{\t}{\xb}{\yb}}
% Draw the second sphere.
\def\Rc{2.4} % sphere radius
\def\angEl{30} % elevation angle
\def\xc{0} % x coordinate of center
\def\yc{0} % y coordinate of center
\def\zc{0} % z coordinate of center
\filldraw[shift={(\xc, \yc)}][ball color= red]
   (0,0,0) circle (\Rc);
\foreach \t in {-60,-20,...,80} {
   \DrawLatitudeCircle[\Rc]{\t}{\xc}{\yc}}
\foreach \t in {-5,-45,...,-175} {
   \DrawLongitudeCircle[\Rc]{\t}{\xc}{\yc}}
\drawIntersectionOfSpheres{\xc}{\yc}{\zc}{\xb}
   {\yb}{\zb}{\Rc}{\Rb}{yellow}
\end{tikzpicture}
```

This code also requires the helper functions for drawing spheres, given in appendix C. Place them just before `\begin{document}`.

The following are additional helper functions to the main function that draws the intersection of the two spheres, `\drawIntersectionOfSpheres`.

```
\newcommand\calculateCenterSpan[4]{
  \pgfmathsetmacro#1{((#2)^2+(#3)^2+(#4)^2)^(1/2)}}
\newcommand\calculateAngtheta[3]{
  \pgfmathsetmacro#1{acos(#2/#3)}}
\newcommand\calculateShiftDistance[4]{
 \pgfmathsetmacro#1{((((#2)^2-(#3)^2)/(2*#4)+#4/2))}}
\newcommand\calculateShiftDistancePercent[3]{
  \pgfmathsetmacro#1{#2/#3}}
\newcommand\calculateCircleRadius[3]{
  \pgfmathsetmacro#1{((#2)^2 - (#3)^2)^(1/2)}
}
% The function below does not use the
% rotation matrices about coordinate axes,
% but instead computes the vector that we
% want to rotate around, then the
% corresponding rotation matrix, and then
% (as before) projects to the xy-plane.
% It also shifts to the correct location.
\newcommand\drawIntersectionOfSpheres[9]{
% Parameters are:  CenterSphere1x, CenterSphere1y,
% CenterSphere1z, CenterSphere2x,
% CenterSphere2y, CenterSphere2z,
% RadiusSphere1, RadiusSphere2, DrawColor.
\pgfmathsetmacro\xchange{#4 - #1}
\pgfmathsetmacro\ychange{#5 - #2}
\pgfmathsetmacro\zchange{#6 - #3}
\pgfmathsetmacro\firstSphereCenterx{#1}
\pgfmathsetmacro\firstSphereCentery{#2}
\calculateCenterSpan\centerSpan{
   \xchange}{\ychange}{\zchange}
\calculateAngtheta\angtheta{\zchange}{\centerSpan}
\calculateShiftDistance\shiftDistance{
   #7}{#8}{\centerSpan}
\calculateShiftDistancePercent\shiftDistancePercent{
   \shiftDistance}{\centerSpan}
\calculateCircleRadius\circleRadius{
   #7}{\shiftDistance}
\pgfmathsetmacro\ux{\ychange}
\pgfmathsetmacro\uy{-\xchange}
\pgfmathsetmacro\C{1-cos(\angtheta)}
\pgfmathsetmacro\L{((\ux)^2 + (\uy)^2)^(1/2)}
\pgfmathsetmacro\first{
  cos(\angtheta) + (\ux)^2*\C/(\L^2)}
\pgfmathsetmacro\second{\ux*\uy*\C/(\L^2)}
\pgfmathsetmacro\third{\ux*\uy*\C/(\L^2)}
\pgfmathsetmacro\fourth{
  cos(\angtheta)+(\uy)^2*\C/(\L^2)}
\tikzset{xyplane/.estyle={cm={\first,\second,\third,
  \fourth,(\firstSphereCenterx+\shiftDistancePercent
  *\xchange,\firstSphereCentery
            +\shiftDistancePercent*\ychange)}}}
\pgfmathsetmacro\dAng{atan(\xchange/\ychange)}
\draw[xyplane, color=#9, ultra thick]
  (-\dAng: \circleRadius) arc
  (-\dAng:-\dAng-180:\circleRadius);
\draw[xyplane, color=#9, ultra thick, dashed]
  (-\dAng+180: \circleRadius) arc
  (-\dAng+180:-\dAng:\circleRadius);
}
```

References

[1] Till Tantau. PGF/Ti*k*Z manual, version 2.10. `http://mirror.ctan.org/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf`, 2010.

[2] Andrew Mertz and William Slough. Graphics with PGF and Ti*k*Z. *TUGboat* 28:1, Proceedings of the Practical TEX 2006 Conference, `http://tug.org/TUGboat/tb28-1/tb88mertz.pdf`, 2007.

[3] Tomasz M. Trzeciak. `http://www.texample.net/tikz/examples/map-projections/`, 2008.

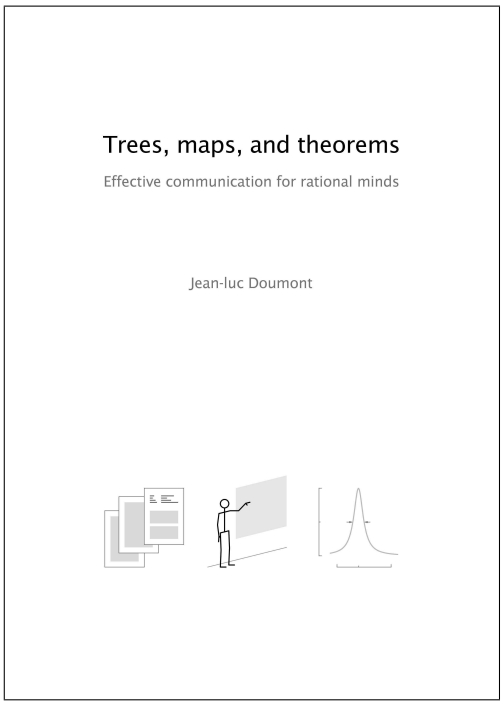
[4] Wikipedia, `http://en.wikipedia.org/wiki/Rotation_matrix`.

◇ Keith Wolcott
Dept. of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920-3099 USA
`kwolcott (at) eiu dot edu`

Book review: *Trees, maps, and theorems*

Pavneet Arora

Jean-luc Doumont, *Trees, maps, and theorems*. Principiae, Kraainem, Belgium. 192pp. Hardcover, $96.00, ISBN 978-90-813677-07.



If a simple sentence is but a message given shape and cadence by syntax, then what of the larger vessels into which it is poured: reports, presentations, and graphics? What principles guide these larger containers of information so that the messages found within aggregate meaningfully, and are made accessible to the audience? This is the theme of Jean-luc Doumont's book *Trees, maps, and theorems: Effective communication for rational minds*.

To frame the discussion, Doumont begins with an unambiguous listing of what he feels to be the *laws* which define the responsibilities of any author of communications material:

1. You, as the author, are responsible for creating communications which impart a message effectively to your audience. This is quite different from the common practice where communications merely showcase a maximal amount of information, and that quite often in a fractured way — the expectation being that it is the responsibility of the audience to rise up and somehow penetrate the author's erudition.
2. Often, communications must compete with noise which impedes the message. You, as the author, must not only anticipate this, but must also work to ensure that the message in your communications is resilient enough to get through.
3. Use redundancy to ensure that the message does get through. That is, that even if the message is made strong, by using redundancy you increase the chances that it will reach your audience.

The emphasis on adapting the communications' form to the recipient is particularly useful, and takes any exposition of design from the abstract and inanimate into the realm of a proper use-case. Here one can test a design by gauging how effectively the message reaches the audience. To take a small example, think of the distinction between a morality play and judicial judgement comprising its opinions and dissents. Both may concern themselves with ethics, but the audience for each is quite distinct. Knowing the audience ensures that the message comes across even in harsh environments.

Beginning with these fundamentals, the book then progresses with chapters covering the three forms of communications it concerns itself with: written documents, oral presentations, and graphical displays. For each of the forms, the discussion is then broken into sections covering the different phases: planning, designing, and constructing.

One may dive into a chapter covering a specific form without losing much by skipping over the discussions about other forms, but within that chapter it is advisable to work through the material linearly. A navigational map, re-introduced with each chapter and updated at each section, makes it easy to understand where one is within the discussion.

The page design has a consistent layout, used throughout, which is described at the outset (a sample spread is at the end of this review). It consists of four distinct areas on each double-page spread:

- main discussion
- illustrations and comments
- frequently asked questions
- common shortcomings and practical advice

With this consistency comes familiarity as one makes one's way through the book. New concepts, as they are introduced, may readily be plugged into a mental map prepared to receive them.

The final chapter animates the guidelines set forth earlier through illustrative examples. Each example comes replete with common errors that are excised as the sample communication undergoes a transformative process and is made better.

So is the book an exemplar of the very principles it espouses? The simple answer is yes. Its emphasis on message over raw information permeates the text.

Illustrations are used to give context to the discussion and are used as needed without distraction.

Within the page design, a grid is used to align elements: textual, tabular, and graphical. The spacing in and around elements makes it easy to navigate, to move back and forth between different levels of abstraction, as well as to return to exceptions without interrupting the flow of the main message.

An attentive reader will notice a compulsive adherence to paragraph layout as the last sentence of each paragraph ends up perfectly aligned at the right margin. While this is admirable, it does produce a disconcerting effect in that one can never be sure if, at the bottom of the page, the paragraph is complete.

As typesetters, both amateur and professional, we pride ourselves in our precision. Sloppy typesetting is considered to be a contraindicator of precise thinking. What Doumont does is to raise our sights higher. While precision is essential in the layout of information, it is a weak standard when it comes to assessing the effectiveness of the underlying message. Giving shape to the message so that it reaches its audience is that real purpose of all communications.

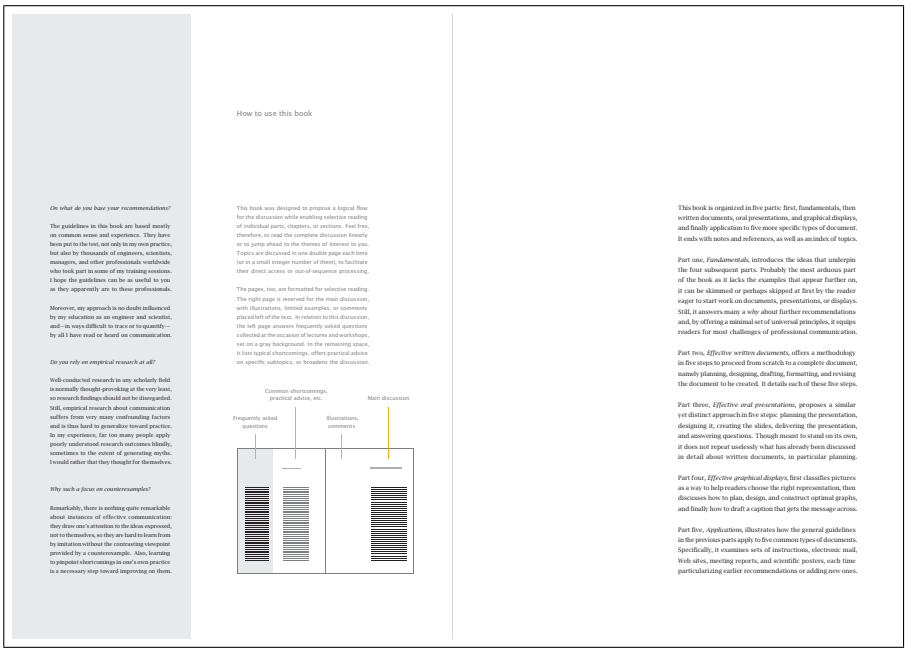
What Doumont illuminates so effectively is that, by taking communications from the mere aesthetic into the realm of signals, we can distill best practises to ensure these signals reach their destination intact and with greater frequency.

If there is an omission in this book, it might be this: I was taught long ago in some sales training that the most effective form of communications is a story. Stories are universal, and often resonate well even across cultures. As such, messages can be made more effective if woven into a narrative. I hope that Doumont will consider this as a request for a follow-up edition which includes this aspect of enhancing signal strength, or as a book on its own that might use this as a theme.

So let me end with a story of my own. I have on my bookshelf a book given to me some decades back during one of my last visits with my grandfather. The book is *Write Better, Speak Better* which was published in 1973 by Reader's Digest. In my arrogance, a not uncommon crutch to the awkwardness of youth, I accepted it glibly and promptly put it aside. By then I had discovered that my affinity was for things technical, and language seemed like a murky inkwell. It was only much later that I came to the realization that it is with the power of language that we can express our ideas — technical and otherwise — most effectively. And because of the satisfaction derived when doing so, I have been playing catch up ever since. Doumont's book is a most worthy addition for anyone interested in expressing themselves to a wider audience, and I recommend it heartily.

◇ Pavneet Arora
Caledon, Ontario, Canada
pavneet_arora (at) bansisworld dot org
http://blog.bansisworld.org/

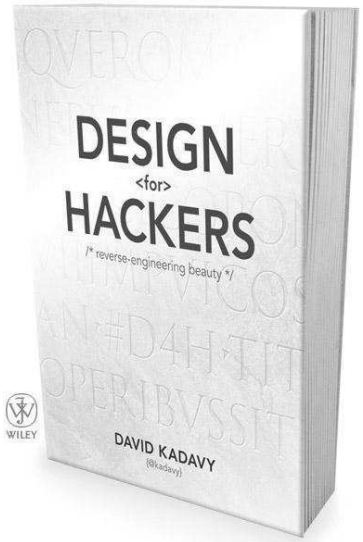
Editor's note: We are happy to note that TEX Users Group members in good standing enjoy an exclusive discount on this book. The details can be found at https://www.tug.org/members.



Book review: *Design for Hackers*

Boris Veytsman

David Kadavy, *Design for Hackers. Reverse-Engineering Beauty*. Wiley & Sons, 2011. 352 pp., Paperback, US39.99. ISBN 978-1-119-99895-2.



In the beginning of the book David Kadavy thanks the Wiley acquisition editor Chris Webb for taking up this project, which another publisher dismissed as "ambitious". After reading the book I could appreciate this story. In 352 pages, the author describes the principles and history of design — the subject that usually takes several semesters of college level courses. If *this* is not ambitious, then what is?

Of course, the book is not intended for a systematic study — rather it is a very popular introduction to whet the appetite of novices and, as Kadavy says, *to provide you with a new set of eyes through which you can see the world anew.* It is clearly intended for the "new media creators": web designers, programmers, management of high-tech companies and other people with a good knowledge of computers, but much less of design and arts. Even if they themselves will not become designers after reading the book, they might better appreciate the job of their designer departments. This in itself would be a very good result. Let us see whether Kadavy achieves this implicit goal.

The book has nine chapters: (1) Why Design Matters, (2) The Purpose of Design, (3) Medium and Form in Typography, (4) Technology and Culture, (5) Fool's Golden Ratio: Understanding Proportions, (6) Holding the Eye: Composition and Design Principles, (7) Enlivening Information: Establishing a Visual Hierarchy, (8) Color Science, and (9) Color Theory. It also includes an unnumbered Introduction and two appendices: (A) Choosing and Pairing Fonts, and (B) Typographic Etiquette.

Simply from this list of chapters a reader can see that the book discusses in detail typography and fonts. It is not surprising, since David Kadavy taught a college course on typography, and seems to be genuinely interested in the subject. Thus TUG members who share this interest might enjoy and appreciate the book.

In my reading, the pages about typesetting are among the best in *Design for Hackers*; the author is very knowledgeable and knows how to convey his knowledge to the reader. Since the book is written mostly about web design, it talks in length about suitability of various fonts and font features for screen rendering. *Design for Hackers* has several fascinating asides: why the Romain du Roi letters commissioned by Louis XIV and designed on a "scientific" grid inspired many digital fonts, the story behind the (in)famous Comic Sans, the design principles of the Georgia family, and many others. These stories alone make the book worth reading.

The chapters about composition and colors are also well-written and quite interesting. The author teaches these subjects by examples: he takes logos, paintings, web pages and explains why they are successful. This exposition is convincing and might be quite revealing for many novices or even experienced readers. For example, in Chapter 6 the author explains how the details of a classical sculpture or a Seurat painting subtly guide the eye to return to the center of the piece. Then he takes a modern logo and shows the same principles in a quite different situation. Kadavy's attention to the details and the skill in revealing these details for the reader are superb.

Kadavy clearly and lucidly explains many rather difficult topics, such as grid design for books and web pages, coordinating font sizes, relationship between colors, color spaces and printing technology, and many, many others.

I would be happy to stop here and congratulate the author and his readers with a very good introductory text on design. Unfortunately, while the publisher provided a bold and dedicated acquisition editor for the book, he seemed to forget about the need for copyediting. *Design for Hackers* has many paragraphs which an experienced copyeditor would delete or demand be rewritten. Following are just a few examples of such paragraphs.

Kadavy twice (p. 3 and p. 46) states that before Gutenberg made books cheap, literacy rates were very low, and few people outside the clergy could read. First, this statement probably does not deserve repeating, and second, it is not quite true. The author himself talks at length about wall graffiti in Pompeii, which demonstrates that in the ancient Roman empire the literacy level was relatively high.

Talking about technology and culture, the author spends several pages on the history of Impressionism and *Salon des Refusés*. The author makes a connection between the new artistic school and the growth of the middle class. A box on page 82 summarizes the take-home message as *REMEMBER: In order for a piece of art or design to really be relevant and important it has to be sensitive to the technological and cultural factors present within the world in which the piece is created. Doing otherwise will result only in the creation of a veneer.* Does this jejune remark really deserve its prominent place? Immediately after this story Kadavy talks about Web 2.0. What does the author want to say? Is Web 2.0 comparable to Impressionism, and if yes, how? Which social changes are relevant to Web 2.0? Maybe Web 2.0 is, using the author's terms, just a veneer?

One of the most prominent examples of the lack of copyediting is the section *SEO Is Design*. Nowhere in the body of the book does the author explain what SEO is, and this reviewer was quite baffled by the passionate pleas in the book about the importance of this enigmatic entity. Only in the index is this acronym spelled out as "Search Engine Optimization": a series of tricks to make your web page go to the top of Google search output. A good editor would likely catch this. A copyeditor could also tell the author that naming two consecutive chapters *Color Science* and *Color Theory* is probably not a good idea.

Another thing a good copyeditor could do is slightly brush up the style of the author. Kadavy's writing is generally easy to read. However, sometimes his colloquial style becomes too flippant, and the result looks rather artificial. For example, Kadavy argues that the "golden" ratio is overused and overhyped, and some other fractions like 2:3 and 3:4 should be used in the design as well. This is fine, but should he really name the corresponding section *Fool's Golden Ratio?* Also, sometimes the author veers into sales speak (*The corporations pay big bucks for ...*).

Since this book is about design, the design of the book itself should be mentioned. It seems to be highly influenced by web pages rather than by traditional typography: ragged right typesetting, paragraphs separated by vertical white space, lots of colored illustrations, etc. Since the author is without doubt a good designer, the result is surprisingly clean and brisk. Marginal captions for figures look excellent, the colors are selected with great care, and the overall impression is striking. The book is printed on glossy paper (probably necessary because of many color illustrations) and is tastefully typeset in beautiful Adobe Garamond. It is a pleasure to open and read.

Returning to the question in the beginning of the review, I can say that Kadavy has written a good introduction to design, interesting for novices and giving food for thought to more experienced readers. I would only wish that he had given the same care and attention to the details of his text as to the design and typesetting of the book.

◇ Boris Veytsman
Computational Materials Science Center, MS 6A2
George Mason University
Fairfax, VA 22030 USA
borisv (at) lk dot net
http://borisv.lk.net

Book review: *Companion to the Papers of Donald Knuth*

David Walden

Donald E. Knuth, *Companion to the Papers of Donald Knuth*. Center for the Study of Language and Information, Stanford, 2012. 441+xiii pp. Paperback, US$35.00, ISBN 978-1575866345. Hardcover, US$70.00, ISBN 978-1575866352.

Between 1992 and 2011, CSLI Publications published eight volumes of collected papers by Donald Knuth:

1. *Literate Programming*
2. *Selected Papers on Computer Science*
3. *Digital Typography*
4. *Selected Papers on Analysis of Algorithms*
5. *Selected Papers on Discrete Mathematics*
6. *Selected Papers on Computer Languages*
7. *Selected Papers on Design of Algorithms*
8. *Selected Papers on Fun and Games*

Now CSLI has published a ninth and final volume in the series: *Companion to the Papers of Donald Knuth*.

This final volume does not completely stand alone, as the other books in the series do. Nonetheless, it is quite wonderful in its way.

In the Preface, Knuth justifies creation of the present book and the series and sketches the content of the present book.

The first chapter contains "six dozen" problems Knuth has submitted to various publications over the years. The second chapter contains previously unpublished solutions to a few of these problems. The solutions to the rest of the problems have been previously published, and one can find pointers to those solutions in chapter 20.

The next four short chapters are small pieces that did not appear in the first eight volumes. One of these is Knuth's well-known note on "Teach Calculus with a Big O".

Chapters 7–17 are transcripts of "conversations" between Knuth and Dikran Karagueuzian (i.e., interviews of Knuth by Karagueuzian). Karagueuzian is the publisher of the volumes in this series. The interviews were done in 1996, which was during the period when *Digital Typography* was being prepared for publication and the completion of TEX and METAFONT was in the not-so-distant past. Thus, there is a good deal of discussion in the interviews about Knuth's efforts with typesetting and font design. There is also a good bit about Knuth quite personally. The titles of the interviews are:

- Prizes and Choices
- Printing
- Life
- Printing (Continued)
- Travel
- Why Computer Science?
- Work Habits and Problem Solving
- Getting Started
- Programming and Languages
- AI, Students, Retirement
- Accidents, Planning, Naming

Altogether there are 145 pages of these interviews following the 40 pages of the first six chapters.

In my view, this first part of the book is worth at least half of the book's price. The rest of the book is perhaps more valuable, especially in the domain of information retrieval for posterity.

The last four chapters are Knuth's CV; a comprehensive list of his books and their translations; an annotated list of his papers (in three categories: major journal publications, secondary works such as reviews and letters to editors, and works of limited circulation); an alphabetical index of titles of his papers (mostly included in the nine book series), and a combined index (names, topics, etc.) covering the entire nine volume series. The last two of these chapters take up 220 pages.

These latter four chapters, especially the final two, are Knuth's effort to make his collected papers accessible to future researchers, as well as his effort for completeness and closure on this series of books. (He states that he plans to devote full time to completion of *The Art of Computer Programming*.) Speaking as an owner of the prior eight books in the series, who spent a lot of time searching through them and their individual indexes while researching the content of the 2010 publication of *TEX's 2^5 Anniversary: A Commemorative Collection* (`http://tug.org/store/tug10/`), this overall index is a godsend.

Providing as it does accessibility to the other volumes in Knuth's series of collected papers, I believe every library featuring books on computing should have this *Companion* volume, whether or not they have the other books in the series. Similarly, I think every computing history archive should have a copy. Individuals who already have several of the earlier volumes in the series will find value in this final volume in the series. And people who admire Knuth and have collected a lot of his books will definitely want a copy of this volume.

As with all Knuth's writings and edited presentations and interviews, the non-lists-and-indexes part of the book is fascinating reading. The lists-and-indexes part of the book is up to Knuth's usual perfectionist standards. I am glad I now have this volume on my bookshelf.

◇ David Walden
`http://www.walden-family.com/texland`

The Treasure Chest

This is a list of selected new packages posted to CTAN (`http://ctan.org`) from November 2011 to March 2012, with descriptions based on the announcements and edited for brevity.

Entries are listed alphabetically within CTAN directories. A few entries which the editors subjectively believed to be of especially wide interest or otherwise notable are starred; of course, this is not intended to slight the other contributions.

We hope this column and its companions will help to make CTAN a more accessible resource to the TEX community. Comments are welcome, as always.

◇ Karl Berry
`http://tug.org/ctan.html`

fonts

`amiri` in `fonts`
Classical Arabic typeface, Naskh style, inspired by the Bulaq Press typeface.

`ipaex` in `fonts`
IPA and IPAex fonts, fixed-width Kana and Kanji and variable-width Western characters.

`libertine-legacy` in `fonts`
Last release of Libertine including Type 1 fonts.

`mdsymbol` in `fonts`
Mathematical symbol font, especially for Adobe Myriad Pro.

`thaifonts-arundina` in `fonts/thai`
DejaVu-compatible serif, sans serif, and monospaced Thai fonts.

graphics

`pst-pulley` in `graphics/pstricks/contrib`
Plot different pulleys.

`pst-solarsystem` in `graphics/pstricks/contrib`
Plot visible planets.

`pst-tools` in `graphics/pstricks/contrib`
Helper functions for PSTricks packages.

`reotex` in `graphics/pgf/contrib`
Draw Reo channels and circuits with Ti*k*Z.

`tikzpfeile` in `graphics/pgf/contrib`
Draw all math arrows with Ti*k*Z.

`vocaltract` in `graphics/pstricks/contrib`
Visualization of the vocal tract.

info

`Einfuehrung` in `info/examples`
Examples for the DANTE e.V. book *Einführung in LaTeX*.

`computer-typesetting-using-latex` in `info/russian`
Extensive guide to LaTeX in Russian.

language

`japanese-otf-uptex` in `language/japanese`
upLaTeX support for `japanese-otf`.

`jfontmaps` in `language/japanese`
Font maps and support scripts for handling Kanji font embedding.

`zhmcjk` in `language/chinese`
Set up CJK fonts dynamically via `zhmetrics`.

macros/generic

`gates` in `macros/generic`
Implementing modular and customizable code.

macros/latex/contrib

`aeb_mobile` in `macros/latex/contrib`
Format a PDF for a smartphone.

`apa6` in `macros/latex/contrib`
Format documents in 6th Edition APA style.

`bchart` in `macros/latex/contrib`
Ti*k*Z-based bar charts.

`bibleref-lds` in `macros/latex/contrib`
Extended references to the LDS scriptures.

`bitelist` in `macros/latex/contrib`
Expandable splitting of token lists.

`cookingsymbols` in `macros/latex/contrib`
Symbols for recipes, such as oven and dish glyphs, made with METAFONT.

`copyrightbox` in `macros/latex/contrib`
Put a small amount of text near an image, possibly rotated.

`diagbox` in `macros/latex/contrib`
Making table heads with diagonal lines.

`documentation` in `macros/latex/contrib`
Documenting C, Java, assembler source.

`droit-fr` in `macros/latex/contrib`
Tools for writing a thesis in French law.

`easyfig` in `macros/latex/contrib`
Easy macro to center image with caption and label.

`fixltxhyph` in `macros/latex/contrib`
Hyphenating a word with an emphasized substring.

`flipbook` in `macros/latex/contrib`
Typeset flipbook animations in corners.

`footnoterange` in `macros/latex/contrib`
Support references to ranges of footnotes.

`fullwidth` in `macros/latex/contrib`
Set left and right (and other) margins.

`gamebook` in `macros/latex/contrib`
Typeset gamebooks and other interactive novels.

`gtrcrd` in `macros/latex/contrib`
Adding chords to lyrics.

`hausarbeit-jura` in `macros/latex/contrib`
Writing "juristische Hausarbeiten" (legal essays) at German universities.

`hletter` in `macros/latex/contrib`
Produce letters with logos, scanned signatures, etc.

`issuulinks` in `macros/latex/contrib`
Produce documents with all links externalized.

`kantlipsum` in `macros/latex/contrib`
Random sentences in Kantian style.

`kdgdocs` in `macros/latex/contrib`
Course and thesis classes for Karel de Grote Univ. College.

`lmake` in `macros/latex/contrib`
Simplify writing of lists that fit a pattern.

`logbox` in `macros/latex/contrib`
Do `\showbox` without stopping compilation.

`longnamefilelist` in `macros/latex/contrib`
Align `\listfiles` output containing long names.

`mattex` in `macros/latex/contrib`
Macros and `.m` files to import Matlab variables.

`media9` in `macros/latex/contrib`
Embed interactive Flash and many other multimedia objects in PDF output.

`menukeys` in `macros/latex/contrib`
Format menu sequences, paths and keystrokes.

`messagebubbles` in `macros/latex/contrib`
Display conversations in message bubbles.

`monofill` in `macros/latex/contrib`
Horizontal alignment of plain or monospaced text.

`nameauth` in `macros/latex/contrib`
Name authority macros for consistency and flexibility.

`nicefilelist` in `macros/latex/contrib`
Improvements for `\listfiles`.

`parnotes` in `macros/latex/contrib`
Notes after every paragraph, or elsewhere.

`philosophers-imprint` in `macros/latex/contrib`
Class for the *Philosophers' Imprint* journal.

`romanbar` in `macros/latex/contrib`
Write Roman numbers, or other text, with bars.

`sasnrdisplay` in `macros/latex/contrib`
Typeset SAS or R code and output.

`sepfootnotes` in `macros/latex/contrib`
Support footnotes coming from a separate file.

`tcolorbox` in `macros/latex/contrib`
Colored and framed text boxes with header, possibly split, especially for code examples.

`title` in `macros/latex/contrib`
Simple headers and footers for pages and floats.

`tui` in `macros/latex/contrib`
Thesis style for the University of the Andes, Colombia.

`usebib` in `macros/latex/contrib`
Reusing bibliographic data.

`xcookybooky` in `macros/latex/contrib`
Typesetting long recipes, with pictures.

`xpatch` in `macros/latex/contrib`
Generalize `etoolbox`.

macros/latex/contrib/beamer-contrib

`appendixnumberbeamer` in `m/l/c/beamer-contrib`
Fix frame numbering in `beamer` with an appendix.

macros/latex/contrib/biblatex-contrib

`biblatex-fiwi` in `m/l/c/biblatex-fiwi`
`biblatex` support for German humanities citations, especially in film studies.

`biblatex-luh-ipw` in `m/l/c/biblatex-contrib`
`biblatex` support for Leibniz University Hannover citations.

`geschichtsfrkl` in `m/l/c/geschichtsfrkl`
`biblatex` support for the history department at the University of Freiburg.

macros/luatex

`lvdebug` in `macros/luatex/latex`
Display boxes, glues, kerns and penalties in the PDF output.

macros/plain

`happy4th` in `macros/plain/contrib`
One hundred pages of fireworks.

`hanoi` in `macros/plain/contrib`
Solve the Towers of Hanoi (up to 15 discs), and learn about category codes.

`reverxii` in `macros/plain/contrib`
Playing Reversi in 938 characters.

support

`check-parens` in `support`
Check for mismatched braces, delimiters, etc.

`checkcites` in `support`
Detect undefined or unused references.

`dosepsbin` in `support`
Extract PS/WMF/TIFF from DOS EPS binary files.

`texlive-dummy` in `support/texlive`
Fulfill dependencies of openSUSE TeX Live packages.

`typeoutfileinfo` in `support`
Display information of a LaTeX file via `readprov`.

Eutypon 26–27, October 2011

Eutypon is the journal of the Greek TEX Friends (`http://www.eutypon.gr`).

Siep Kroonenberg, External graphics for LATEX; pp. 1–13

LATEX documents can include many kinds of graphics, ranging from photographs to illustrations, diagrams and data plots. Often, the best and simplest choice is to create the graphic with external software, and save it in or export it to a LATEX-compatible format. This article surveys available options. Note: This is an update of a paper published originally in *MAPS* no. 35 (2007), pp. 18–26. (*Article in English.*)

Georgios Tsalakos, The drawing package PSTricks; pp. 15–27

PSTricks and its add-on ("contributed") packages make a powerful drawing tool for users of LATEX and other TEX-based typesetting systems. In this paper, a summary-through-examples is given of the basic capacities of PSTricks, and the additional capabilities offered by contributed packages. Also, examples are shown for the preparation and the embedding of drawings and images in school reports and other prints. (*Article in Greek with English abstract.*)

Dimitrios Filippou, In the age of the typewriter; pp. 29–59

In 2011, a news item made headlines around the world: "Last typewriter factory left in the world closes its doors". On the occasion of this rather false piece of news, this article presents a brief history of the typewriter, from the first mechanical typewriters to the most modern electronic typewriters, which also tend to disappear. Special reference is made to typewriters with Greek keyboards, which reached their peak of glory in the early 1980s, when university notes and textbooks were massively produced in Greece. (*Article in Greek with English abstract.*)

Apostolos Syropoulos, TEXniques: Looking for certain glyphs in OpenType fonts; pp. 61–62

In this paper, we present a second, revised version of our "Byzantine" music fonts. We also present a new approach for a more efficient use of these fonts with LATEX, and its ancestor TEX. (*Article in Greek with English abstract.*)

Georgios Tsalakos, Book review: H. Voss, *PSTricks*; pp. 63–64

PSTricks — Graphics and PostScript for TEX and LATEX, UIT Cambridge, Cambridge, UK, 2011. (*Article in Greek.*)

[Received from Dimitrios Filippou and Apostolos Syropoulos.]

Die TEXnische Komödie 4/2011–1/2012

Die TEXnische Komödie is the journal of DANTE e.V., the German-language TEX user group (`http://www.dante.de`). [Editorial items are omitted.]

Die TEXnische Komödie 4/2011

Manuel Pégourié-Gonnard, Attribute und Farben [Attributes and colors]; pp. 24–49

We take a look at attributes, a LuaTEX extension, and their use for implementing colors. After explaining the underlying concepts and the TEX and Lua interfaces we provide a short overview of the basics behind the classical color implementation in LATEX and its shortcomings. We then show how these shortcomings can be solved using attributes. We also show that attributes can be used for other needs, not just colors.

Arno Trautmann, Das Paket `chickenize` – Spaß mit Node-Manipulationen in LuaTEX [The `chickenize` package — fun with node manipulations in LuaTEX]; pp. 50–57

[Translation published in this issue of *TUGboat*.]

Dirk Hünniger, Ein Programm zur Konvertierung von Artikeln der Wikipedia nach LATEX [Converting Wikipedia articles to LATEX]; pp. 58–60

For various reasons it would be nice to be able to generate a LATEX file from a Wikipedia article. Manual conversion is elaborate and prone to errors. Therefore a compiler has been developed and made publicly available for Windows and Linux. Furthermore, the source code has been published under an open license. The tool is also used for projects such as Wikibooks.

Petra Rübe-Pugliese, Aktuelle experimentelle deutsche Trennmuster unter Debian-TEX Live [Up-to-date experimental German hyphenation patterns for Debian TEX Live]; pp. 61–65

In this article we explain the installation of the latest German hyphenation patterns on Debian.

Philipp Lehman, Zu den Nachteilen von BibTEX [On the disadvantages of BibTEX]; pp. 66–67

When we discuss the disadvantages of BibTEX there are two main topics:

1. BibTEX and BibTEX8 assume that a character is represented by one byte. This however does not apply to UTF-8 where for certain characters more than one byte is used.

2. It is not just UTF-8 but Unicode in general as the comprehensive standard for encoding characters and many associated topics such as sorting.

Joseph Wright, LaTeX3 Roadmap; pp. 68–70

One of the questions that comes up from time to time is what the 'roadmap' is for LaTeX3 development. While there is not an official plan, I certainly have some ideas on what I'd like to see addressed in a concrete way. This is all rather flexible, but I'll try to outline some areas for attention.

Herbert Müller, Berichtigung zu »Von \pageref zu \hyperpage« [Corrections to the article "From \pageref to \hyperpage"]; pp. 71–72

[Letter to the editor with corrections.]

Die TeXnische Komödie 1/2012

Agnieszka Okońska, LaTeX für Juristen [LaTeX for attorneys]; pp. 6–12

The search for the perfect solution from a user's point of view. While LaTeX is commonly used for creating scientific papers in the natural sciences or in economics its use in other areas of science is still rare. To members of these disciplines LaTeX may offer advantages as well, especially to attorneys.

Herbert Voss, Datumsfunktionen mit LuaTeX [Date functions with LuaTeX]; pp. 13–19

With pdfLaTeX the current date can be printed using the macro \today without any problems; for printing the time, one may use the package datetime (although this package does not supporting printing seconds). With LuaTeX or LuaLaTeX, however, one can use the date- and time routines of Lua, which allow easy formatting of date and time as well.

Herbert Voss, Multilinguale Texte mit LuaLaTeX — ein Versuch [Multilingual texts with LuaLaTeX — An experiment]; pp. 20–27

The TeX flavor Ω (Omega) was developed to overcome LaTeX's restrictions and to allow using 16-bit fonts. The successor of Ω was ℵ (Aleph), which provided a first step to the general use of Unicode. ℵ is not under development any more since with LuaTeX there is a successor that allows multilingual texts in a way that has not been possible with TeX before. In this article the author describes his experience concerning writing multilingual documents with LuaTeX.

Wolfgang Beinert, Typographischer Punkt [The typographical point]; pp. 28–30

Point typographique, Didot-point, pica point, PostScript point ... The "typographical system of units", abbreviated as "point", originally named as a "Point typographique". The typographical point is the smallest unit of a typographical point unit system. It was named in the mid-18th century in France for an asymmetrical system of measurement for uniform sizing of letters, font sizes, and distances.

Heiko Oberdiek and Herbert Voss, Index mit Fortsetzungsanzeige [Index with "continuation" signs]; pp. 31–32

Using fancyhdr or scrpage2 one can print the letter range of the index page in the header. But one has to invest quite a bit of effort if such a letter is a continuation from the previous page.

[Received from Herbert Voß.]

The Asian Journal of TeX, Volumes 4–5 (2010–2011)

The Asian Journal of TeX is the publication of the Korean TeX Society (http://ktug.kr).

***AJT* Volume 4, Number 1**

Dohyun Kim, XeTeX-k*o*: A XeTeX macro package for processing Korean documents; pp. 1–30

XeTeX-k*o* is a macro package for typesetting Korean documents, including old Hangul texts as well, upon the XeTeX engine. XeTeX is a sophisticated TeX engine which supports full Unicode encoding and OpenType layout features. Using XeTeX itself, however, is not fully satisfactory for Korean usage, especially because of the relatively poor quality of Latin/Greek/Cyrillic glyphs in many Korean fonts. For this reason and others, XeTeX-k*o* has been recently developed, mainly focusing on how to typeset with different fonts between Western and Korean characters. This paper presents the current state of the XeTeX-k*o* package along with its main features and usage, including among others how to configure Korean fonts, how to change character spacing, and what to be prepared for in typesetting old Hangul texts. At the end of the paper, some limitations of the current XeTeX-k*o* package will also be discussed as a warning to the users. (Article in Korean.)

Juho Lee, Fontspec: A wing of XeTeX; pp. 31–68

(Article in Korean.)

Hoze Yi, On technical writing of manuals; pp. 69–80

(Article in Korean.)

***AJT* Volume 4, Number 2**

In-Sung Cho, Understanding and using coordinates in PSTricks with application to

plotting functions from economics models; pp. 81–100

The article examines the coordinate systems in PSTricks. Though the default is the Cartesian coordinate system, one can also use the polar coordinate system once the command `\SpecialCoor` is declared. The command `\SpecialCoor` allows greater variety of expression of the Cartesian and polar coordinates, such as PostScript language and predefined nodes. Understanding the features of the various expressions of coordinates can make a tedious graphing job interesting or challenging. The article provides some examples of plotting economics models: markets, monopoly, and Cournot duopoly. (Article in Korean.)

Akira Takahashi, SB portable Japanese TEX environment for Windows; pp. 101–110

This paper describes a method for developing a Japanese TEX environment on a USB drive for Windows users. In order to sufficiently support the Japanese TEX environment, not only W32TEX, which contains upLATEX and TEXworks, but also Ghostscript, which can handle TrueType/OpenType CJK fonts, GSview, and the Perl execution environment are installed. (Article in English.)

Masataka Kaneko and Setsuo Takato, The extension of KETpic functions — Meta commands and their applications; pp. 111–120

Though LATEX has become the standard tool for editing high-quality mathematical documents, the use of graphics in LATEX tends to be unsatisfactory. Also it is desirable that the capability of generating tables and page layout in the preferred style be added to LATEX. The authors have developed KETpic, a computer algebra system (CAS)-based plug-in for high-quality graphics in LATEX documents. In this paper, we will show how newly developed functionality in KETpic can easily generate new environments or graphical commands of LATEX, so that LATEX can be endowed with additional capabilities. (Article in English.)

Shin-ichi Todoroki and Tomoya Konishi, BibTEX-based manuscript writing support system for researchers; pp. 121–128

A list of publications can help researchers with their writing if each item on the list includes links to their manuscript files stored in their personal computers. That way, they can quickly find their previous work, figures and photographs from the list and reduce their writing time by reusing them. We have developed a system providing such lists on a web browser by using Ruby scripts and BibTEX `bib` files. This system is designed to generate an author's list of publications in various formats and to manage current manuscripts with the aim of providing an adequate return on the time spent keeping the database up to date. (Article in English.)

AJT Volume 5, Number 1

Kangsoo Kim, Typesetting a book with the `oblivoir` class; pp. 1–59

(Article in Korean.)

AJT Volume 5, Number 2

Committee of the TEX Conference Japan 2011, Record of the distribution round table; pp. 61–64

The session "Distribution Round Table" was held at the TEX Conference Japan 2011 on 22nd October, 2011. The objective of this session was to introduce Japanese TEX distributors or packagers on different platforms, TEX distribution developers, and TEX engine developers to one another. They discussed many topics, including clarifying what is the difference between Japanese localized TEX environments and the original TEX Live environment. The topic of the `updmap` (update font map) script for mapping Japanese Kanji stood out, and that the changes in Japanese Kanji mappings of `updmap` would be merged in upstream to TEX Live. In other topics, they discussed how many Japanese TEX users typeset documents and preview the resulting DVI/PDF files with or without synchronizing their source TEX files. They also announced that the TEX Live system has a local repository system. (Article in Japanese.)

Hironori Kitagawa, Development of the LuaTEX-ja package; pp. 65–79

LuaTEX-ja is a macro package for typesetting Japanese documents with LuaTEX. It enjoys the improved flexibility of LuaTEX in typesetting TEX documents, so eliminating some unwanted features of pTEX, the widely-used variant of TEX for the Japanese language. In this paper, we describe the specifications, the current status, and some internal processing methods of LuaTEX-ja. (Article in English.)

[Received from Jin-Hwan Cho.]

TEX Consultants

The information here comes from the consultants themselves. We do not include information we know to be false, but we cannot check out any of the information; we are transmitting it to you as it was given to us and do not promise it is correct. Also, this is not an official endorsement of the people listed here. We provide this list to enable you to contact service providers and decide for yourself whether to hire one.

TUG also provides an online list of consultants at `http://tug.org/consultants.html`. If you'd like to be listed, please see that web page.

Aicart Martinez, Mercè
Tarragona 102 4^o 2^a
08015 Barcelona, Spain
+34 932267827
Email: `m.aicart (at) ono.com`
Web: `http://www.edilatex.com`
We provide, at reasonable low cost, LaTeX or TeX page layout and typesetting services to authors or publishers world-wide. We have been in business since the beginning of 1990. For more information visit our web site.

Dangerous Curve
PO Box 532281
Los Angeles, CA 90053
+1 213-617-8483
Email: `typesetting (at) dangerouscurve.org`
Web: `http://dangerouscurve.org/tex.html`
We are your macro specialists for TeX or LaTeX fine typography specs beyond those of the average LaTeX macro package. If you use XeTeX, we are your microtypography specialists. We take special care to typeset mathematics well.

Not that picky? We also handle most of your typical TeX and LaTeX typesetting needs.

We have been typesetting in the commercial and academic worlds since 1979.

Our team includes Masters-level computer scientists, journeyman typographers, graphic designers, letterform/font designers, artists, and a co-author of a TeX book.

Hendrickson, Amy
Brookline, MA, USA
Email: `amyh (at) texnology.com`
Web: `http://www.texnology.com`
LaTeX macro writing our speciality for more than 25 years: macro packages for major publishing companies, author support; journal macros for American Geophysical Union, Proceedings of the National Academy of Sciences, and many more.

Scientific journal and e-journal design and production.

Hendrickson, Amy (cont'd)
LaTeX training, at MIT, Harvard, many more venues. Customized on site training available.

Please visit our site for samples, and get in touch. We are particularly glad to take on adventurous new uses for LaTeX, for instance, web based report generation including graphics, for bioinformatics or other applications.

Latchman, David
4113 Planz Road Apt. C
Bakersfield, CA 93309-5935
+1 518-951-8786
Email: `david.latchman (at) gmail.com`
Web: `http://www.elance.com/s/dlatchman`
Proficient and experienced LaTeX typesetter for books, monographs, journals and papers allowing your documents and books to look their possible best especially with regards to technical documents. Graphics/data rendered either using Ti*k*Z or Gnuplot. Portfolio available on request.

Moody, Trent
1981 Montecito Ave.
Mountain View, CA 94043
+1 650-283-7042
Email: `trent.moody (at) ymail.com`
Construction of technical documents with mathematical content from hand written (or partially formatted) sources. Delivered documents will be `.tex` and `.pdf` files produced with TeX or/and LaTeX. Delivered documents can be publication ready manuscripts, macro libraries for modular document development, or mathematical libraries for document reuse.

I am an independent contractor with a PhD in mathematical physics from the University of California, Santa Cruz.

Peter, Steve
295 N Bridge St.
Somerville, NJ 08876
+1 732 306-6309
Email: `speter (at) mac.com`
Specializing in foreign language, multilingual, linguistic, and technical typesetting using most flavors of TeX, I have typeset books for Pragmatic Programmers, Oxford University Press, Routledge, and Kluwer, among others, and have helped numerous authors turn rough manuscripts, some with dozens of languages, into beautiful camera-ready copy. In addition, I've helped publishers write, maintain, and streamline TeX-based publishing systems. I have an MA in Linguistics from Harvard University and live in the New York metro area.

Shanmugam, R.
No. 38/1 (New No. 65), Veerapandian Nagar, Ist St.
Choolaimedu, Chennai-600094, Tamilnadu, India
+91 9841061058
Email: rshanmugam92 (at) yahoo.com
As a Consultant, I provide consultation, training, and full service support to individuals, authors, typesetters, publishers, organizations, institutions, etc. I support leading BPO/KPO/ITES/Publishing companies in implementing latest technologies with high level automation in the field of Typesetting/Prepress, ePublishing, XML2PAGE, WEBTechnology, DataConversion, Digitization, Cross-media publishing, etc., with highly competitive prices.
I provide consultation in building business models & technology to develop your customer base and community, streamlining processes in getting ROI on our workflow, New business opportunities through improved workflow, Developing eMarketing/E-Business Strategy, etc. I have been in the field BPO/KPO/ITES, Typesetting, and ePublishing for 16 years, handled various projects. I am a software consultant with Master's Degree. I have sound knowledge in TEX, LATEX2$_\varepsilon$, XMLTEX, Quark, InDesign, XML, MathML, DTD, XSLT, XSL-FO, Schema, ebooks, OeB, etc.

Sievers, Martin
Im Treff 8, 54296 Trier, Germany
+49 651 4936567-0
Email: info (at) schoenerpublizieren.com
Web: http://www.schoenerpublizieren.com
As a mathematician with ten years of typesetting experience I offer TEX and LATEX services and consulting for the whole academic sector (individuals, universities, publishers) and everybody looking for a high-quality output of his documents.

From setting up entire book projects to last-minute help, from creating individual templates, packages and citation styles (BIBTEX, biblatex) to typesetting your math, tables or graphics — just contact me with information on your project.

Sofka, Michael
8 Providence St.
Albany, NY 12203
+1 518 331-3457
Email: michael.sofka (at) gmail.com
Skilled, personalized TEX and LATEX consulting and programming services.

I offer over 25 years of experience in programming, macro writing, and typesetting books, articles, newsletters, and theses in TEXand LATEX: Automated document conversion; Programming in Perl, C, C++ and other languages; Writing and customizing macro packages in TEX or LATEX; Generating custom output in PDF, HTML and XML; Data format conversion; Databases.

If you have a specialized TEX or LATEX need, or if you are looking for the solution to your typographic problems, contact me. I will be happy to discuss your project.

Veytsman, Boris
46871 Antioch Pl.
Sterling, VA 20164
+1 703 915-2406
Email: borisv (at) lk.net
Web: http://www.borisv.lk.net
TEX and LATEX consulting, training and seminars. Integration with databases, automated document preparation, custom LATEX packages, conversions and much more. I have about seventeen years of experience in TEX and thirty years of experience in teaching & training. I have authored several packages on CTAN, published papers in TEX related journals, and conducted several workshops on TEX and related subjects.

TUG Institutional Members

American Mathematical Society,
Providence, *Rhode Island*

Aware Software, Inc.,
Midland Park, *New Jersey*

Banca d'Italia,
Roma, *Italy*

Center for Computing Sciences,
Bowie, *Maryland*

Certicom Corp.,
Mississauga, *Ontario*, *Canada*

CSTUG, *Praha*, *Czech Republic*

diacriTech, *Chennai*, *India*

Florida State University,
School of Computational Science and Information Technology,
Tallahassee, *Florida*

IBM Corporation,
T J Watson Research Center,
Yorktown, *New York*

Institute for Defense Analyses,
Center for Communications Research,
Princeton, *New Jersey*

LAMFA CNRS UMR 6140,
Amiens, *France*

MacKichan Software, Inc.,
Washington/New Mexico, *USA*

Marquette University,
Department of Mathematics, Statistics and Computer Science,
Milwaukee, *Wisconsin*

Masaryk University,
Faculty of Informatics,
Brno, *Czech Republic*

MOSEK ApS,
Copenhagen, *Denmark*

New York University,
Academic Computing Facility,
New York, *New York*

Springer-Verlag Heidelberg,
Heidelberg, *Germany*

StackExchange,
New York City, *New York*

Stanford University,
Computer Science Department,
Stanford, *California*

Stockholm University,
Department of Mathematics,
Stockholm, *Sweden*

University College, Cork,
Computer Centre,
Cork, *Ireland*

Université Laval,
Ste-Foy, *Québec*, *Canada*

University of Ontario,
Institute of Technology,
Oshawa, *Ontario*, *Canada*

University of Oslo,
Institute of Informatics,
Blindern, *Oslo*, *Norway*

TUG financial statements for 2011

Karl Berry

The financial statements for 2011 have been reviewed by the TUG board but have not been audited. As a US tax-exempt organization, TUG's annual information returns are publicly available on our web site: `http://www.tug.org/tax-exempt`.

Revenue (income) highlights

Membership dues revenue was down about 3% in 2011 compared to 2010, while product sales and advertising were slightly up. Contributions were also up: we received about $1200 in donations from individuals just in December, and $3220 from River Valley Technologies, from the TUG'11 conference. (Thank you to all!) Overall, 2011 income was down 1%.

Cost of Goods Sold and Expenses highlights, and the bottom line

Payroll, office expenses, and *TUGboat* and DVD production and mailing continue to be the major expense items. All were nearly as budgeted for 2011; overall, 2011 expenses were up about 2% from 2010.

Often we have a prior year adjustment that takes place early in the year to compensate for something that had to be estimated at the time the books were closed at year end; in 2011 the total of such adjustments was positive for the bottom line: $918.

Despite the slight drop in income and slight increase in expenses, the net result for the year was still positive: about $4,000.

Balance sheet highlights

TUG's end-of-year asset level is up around $5,000 (3%) in 2011 compared to 2010.

The Committed Funds are administered by TUG specifically for designated projects: LaTeX3, the TeX development fund, CTAN, and so forth. Incoming donations have been allocated accordingly and are disbursed as the projects progress. TUG charges no overhead for administering these funds.

The Prepaid Member Income category is member dues that were paid in earlier years for the current year (and beyond). Most of this liability (the 2012 portion) was converted into regular Membership Dues in January of 2012.

The payroll liabilities are for 2011 state and federal taxes due January 15, 2012.

Summary

TUG remains financially solid as we enter 2012. Membership fees remain unchanged in 2012; the last increase was in 2010.

TUG continues to work closely with the other TeX user groups and ad hoc committees on many activities to benefit the TeX community.

TUG 12/31/2011 (vs. 2010) Revenue and Expenses

	Jan - Dec 11	Jan - Dec 10
Ordinary Income/Expense		
Income		
Membership Dues	101,160	104,261
Product Sales	5,056	4,224
Contributions Income	7,206	6,515
Annual Conference	3,220	2,820
Interest Income	882	1,416
Advertising Income	545	265
Total Income	118,069	119,501
Cost of Goods Sold		
TUGboat Prod/Mailing	24,774	24,001
Software Production/Mailing	2,710	3,055
Postage/Delivery - Members	1,795	2,149
Lucida Open Type Font Project	1,430	
Member Renewal	458	523
Copy/Printing for members		47
Total COGS	31,167	29,775
Gross Profit	86,902	89,726
Expense		
Contributions made by TUG	2,000	2,000
Office Overhead	12,219	12,161
Payroll Exp	66,572	65,778
Lucida OpenType Development	1,250	500
Depreciation Expense		260
Total Expense	82,041	80,699
Net Ordinary Income	4,861	9,027
Other Income/Expense		
Other Income		
Prior year adjust	-1,726	1,969
Total Other Income	-1,726	1,969
Net Income	3,135	10,996

TUG 12/31/2011 (vs. 2010) Balance Sheet

	Dec 31, 11	Dec 31, 10
ASSETS		
Current Assets		
Total Checking/Savings	185,696	180,673
Accounts Receivable	345	285
Total Current Assets	186,041	180,958
Fixed Assets		808
TOTAL ASSETS	186,041	181,766
LIABILITIES & EQUITY		
Liabilities		
Total Committed Funds	43,762	41,405
Total TUG conference	-2,650	
Total Prepaid member income	4,645	3,160
Total Payroll Liabilities	1,035	1,087
Total Current Liabilities	46,792	45,652
TOTAL LIABILITIES	46,792	45,652
Equity		
Unrestricted	136,114	125,117
Net Income	3,135	10,997
Total Equity	139,249	136,114
TOTAL LIABILITIES & EQUITY	186,041	181,766

◇ Karl Berry
TUG treasurer
http://tug.org/tax-exempt

Calendar

2012

Apr 5 – 6 TYPO San Francisco, "It's all about Connections", sponsored by FontShop. www.atypi.org/events/typo-san-francisco

Apr 14 GUTenberg Journée et Assemblée Générale, Paris, France. www.gutenberg.eu.org/Journee-GUTenberg-2012

Apr 29 – May 3 BachoTEX 2012: 20th BachoTEX Conference, "Twenty years after", Bachotek, Poland. www.gust.org.pl/bachotex/2012

May 1 **TUG 2012** presentation proposal deadline. tug.org/tug2012

May 15 **TUG 2012** early bird registration deadline. tug.org/tug2012

Jun 1 **TUG 2012** preprints deadline. tug.org/tug2012

Jun 4 – Jul 27 Rare Book School, University of Virginia, Charlottesville, Virginia. Many one-week courses on type, bookmaking, printing, and related topics. www.rarebookschool.org/schedule

Jun 15 – 18 ISType: Istanbul Typography Seminars, Istanbul, Turkey. istype.com

Jun 17 Pinting Arts Fair, Museum of Printing, North Andover, Massachusetts. www.museumofprinting.org/txp/events2

Jun 19 – 22 Book history workshop, École de l'institut d'histoire du livre, Lyon, France. ihl.enssib.fr

Jun 26 – 29 SHARP 2012, "The Battle for Books", Society for the History of Authorship, Reading & Publishing. Dublin, Ireland. www.sharpweb.org

Jun 30 – Jul 1 The Tenth International Conference on the Book, Universidad Abat Otiba CEU, Barcelona, Spain. booksandpublishing.com/conference-2012

Jul 8 – 28 Wells College Book Arts Center, Summer Institute, Aurora, New York. Three one-week sessions. www.wells.edu/academics/programs/book-arts

Jul 9 – 13 "Towards a Digital Mathematics Library" (DML 2012), Bremen, Germany. www.fi.muni.cz/~sojka/dml-2012.html

---

TUG 2012
Boston, Massachusetts.

Jul 16 LaTeX workshop (concurrent)

Jul 16 – 18 The 33rd annual meeting of the TEX Users Group. Presentations covering the TEX world. tug.org/tug2012

Jul 16 – 22 Digital Humanities 2012, Alliance of Digital Humanities Organizations, University of Hamburg, Germany. www.digitalhumanities.org/conference

Jul 30 **TUG 2012** final papers deadline.

Jul 31 – Aug 5 TypeCon 2012: "MKE SHFT", Milwaukee, Wisconsin. www.typecon.com

Aug 5 – 9 SIGGRAPH 2012, Los Angeles, California. s2012.siggraph.org

Aug 23 – 26 TEXperience 2012 (5th TEXperience Conference, organized by CSTUG), Morávka, The Czech Republic. katedry.osu.cz/kma/TeXperience2012

Sep 4 – 7 ACM Symposium on Document Engineering, Paris, France doceng2012.wp.institut-telecom.fr

Sep 16 – 21 XML Summer School, St Edmund Hall, Oxford University, Oxford, UK. www.xmlsummerschool.com

Oct 8 – 12 EuroTEX 2012, 6th International ConTEXt user meeting, and DANTE Herbsttagung and 47th meeting, Breskens, The Netherlands. meeting.contextgarden.net/2012

Oct 10 – 14 Association Typographique Internationale (ATypI) annual conference, Hong Kong. www.atypi.org

Oct 12 – 13 American Printing History Association's 37th annual conference, "At the Crossroads: Living Letterform Traditions", Columbia College Chicago, Illinois. www.printinghistory.org/about/calendar.php

Status as of 15 March 2012

For additional information on TUG-sponsored events listed here, contact the TUG office (+1 503 223-9994, fax: +1 206 203-3960, e-mail: office@tug.org). For events sponsored by other organizations, please use the contact address provided.

A combined calendar for all user groups is online at texcalendar.dante.de.

Other calendars of typographic interest are linked from tug.org/calendar.html.

TUGBOAT Volume 33 (2012), No. 1