TUGboat

Volume 40, Number 2 / 2019
TUG 2019 Conference Proceedings

TeX Users Group

TUGboat (ISSN 0896-3207) is published by the TeX Users Group. Web: `tug.org/TUGboat`.

Individual memberships

2019 dues for individual members are as follows:

- Trial rate for new members: $20.
- Regular members: $105.
- Special rate: $75.

The special rate is available to students, seniors, and citizens of countries with modest economies, as detailed on our web site. Members may also choose to receive *TUGboat* and other benefits electronically, at a discount. All membership options are described at `tug.org/join.html`.

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 carries with it such rights and responsibilities as voting in TUG elections. All the details are on the TUG web site.

Journal subscriptions

TUGboat subscriptions (non-voting) are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. The subscription rate for 2019 is $110.

Institutional memberships

Institutional membership is primarily a means of showing continuing interest in and support for TeX and TUG. It also provides a discounted membership rate, site-wide electronic access, and other benefits. For further information, see `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.

[printing date: September 2019]

Printed in U.S.A.

Board of Directors

Addresses

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

Telephone

+1 503 223-9994

Fax

+1 815 301-3568

Web

`tug.org`
`tug.org/TUGboat`

Electronic Mail

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`

2019 Conference Proceedings

TeX Users Group
Fortieth annual TUG meeting
Palo Alto, California, USA
August 9–11, 2019

TUGboat

COMMUNICATIONS OF THE TeX USERS GROUP

TUGboat Editor Barbara Beeton
Proceedings Editor Karl Berry

Volume 40, Number 2, 2019
Portland, Oregon, U.S.A.

TUG 2019 — Palo Alto, California, USA

https://tug.org/2019
tug2019@tug.org

Sheraton Palo Alto Hotel
625 El Camino Real
Palo Alto, CA 94301

Sponsors

TEX Users Group ▪ DANTE e.V.

Adobe ▪ Google ▪ Overleaf ▪ Pearson ▪ STM Document Engineering Pvt Ltd
with notable assistance from individual contributors. *Thanks to all!*

Special guest **Donald E. Knuth**

Conference committee

Karl Berry ▪ Jennifer Claudio ▪ Robin Laakso ▪ Boris Veytsman

Participants

Amine Anane, Montreal, Canada
William Adams, Mechanicsburg, PA
Pavneet Arora, Bolton, ON
Takuto Asakura, National Institute of Informatics, Japan
Brian Bartling, AMS
Nelson Beebe, University of Utah
Barbara Beeton, TEX Users Group
Antoine Bossard, Kanagawa University
Erik Braun, CTAN
Aleksandar Bradic, Supplyframe
Kevin Edward Cain, Institute for the Study of Graphical Heritage
Jaeyoung Choi, Soongsil University
Dennis Claudio
Jennifer Claudio, Oak Grove High School
Alan Davis, Oakland, CA
Susan DeMeritt, IDA/CCR, La Jolla, CA
David Fuchs
Shawn Gaither, Adobe
Federico Garcia-De Castro, Alia Musica Pittsburgh
Peter Giunta, Somerville, MA
Paul Gill, Los Gatos, CA
Steve Grathwohl, Chapel Hill, NC
Sally Ha, Oak Grove High School
Matthew Hardy, Adobe
Jim Hefferon, St Michael's College
Joe Hogg, Los Angeles, CA
Chris Jimenez, Stetson University
Douglas Johnson, Savannah, GA
Shakthi Kannan, Chennai, India
Patrick Kelley, Google
Rohit Khare, Google
Donald Knuth, Stanford University
Richard Koch, University of Oregon
Yusuke Kuroki, Yokohama, Japan
Robin Laakso, TEX Users Group
Doug McKenna, Mathemaesthetics, Inc.
Henri Menke, University of Otago
Frank Mittelbach, LATEX3 Project
Mostafa Mortezaie, DeVry University
Andrea O'Riordan, UCLA
Behrooz Parhami, UC Santa Barbara
Emily Park, San Jose, CA
Ganesh Pimpale, San Jose, CA
Cheryl Ponchin, IDA/CCR, Princeton, NJ
Arthur Reutenauer, Uppsala, SE
Rishi T, STM Document Engineering Pvt Ltd
Tomas Rokicki, Palo Alto, CA
Chris Rowley, LATEX3 Project
Martin Ruckert, Hochschule München
Edgaras Šakuras, VTEX
Herbert Schulz, Naperville, IL
Senthil, San Ramon, CA
Michael Sharpe, UCSD
Keiichiro Shikano, Tokyo
Shreevatsa R, Sunnyvale, CA
Ondřej Sojka, CSTUG
Petr Sojka, Masaryk University, Faculty of Informatics and CSTUG
Nate Stemen, Overleaf
Paulo Ney de Souza, BooksInBytes
Linas Stonys, VTEX
Yusuke Terada, Tokyo Educational Institute
Rebecca Turner, Google
Brian Tracy, Brown University
Didier Verna, EPITA
Boris Veytsman, Chan Zuckerberg Initiative and George Mason University
Paul Vojta, UC Berkeley
Joseph Weening, San Diego, CA
Alan Wetmore, Silver Spring, MD
Uwe Ziegenhagen, Cologne, Germany
Jiří Zlatuška, Masaryk University, Faculty of Informatics

TUG 2019 program

(* = presenter)

Friday August 9	8:15 am	*registration*	
	8:55 am	Boris Veytsman, TeX Users Group	*Welcome*
	9:00 am	Erik Braun, CTAN	*Current state of CTAN*
	9:30 am	Arthur Reutenauer, Uppsala, Sweden	*The state of XeTeX*
	10:00 am	Frank Mittelbach, LaTeX3 Project	*The LaTeX "dev" format*
	10:30 am	*break*	
	10:45 am	Frank Mittelbach	*Taming UTF-8 in pdfTeX*
	11:15 am	Uwe Ziegenhagen, Cologne, Germany	*Combining LaTeX with Python*
	11:45 am	Henri Menke, University of Otago	*Parsing complex data formats in LuaTeX with LPEG*
	12:15 pm	*lunch*	
	1:30 pm	Dick Koch, University of Oregon	*Big changes in MacTeX, and why users should never notice*
	1:45 pm	Nate Stemen, Overleaf, Inc.	*A few words about Overleaf*
	2:00 pm	Aravind Rajendran, Rishikesan Nair T*, Rajagopal C.V., STM Doc. Eng. Pvt Ltd	*Neptune — a proofing framework for LaTeX authors*
	2:30 pm	Pavneet Arora, Bolton, ON	*Rain Rain Go Away: Some thoughts on rain protection, and its uses*
	3:00 pm	*break*	
	3:15 pm	Shreevatsa R, Sunnyvale, CA	*What I learned from trying to read the TeX program*
	3:45 pm	Petr Sojka, Masaryk University & CSTUG	*TeX in Schools? Just Say Yes, given that ...*
	4:15 pm	Shakthi Kannan, Chennai, India	*XeTeX Book Template*
	4:45 pm	Jim Hefferon, St Michael's College	*What do today's newcomers want?*
	5:15 pm	*TUG Annual General Meeting*	
Saturday August 10	8:55 am	*announcements*	
	9:00 am	Petr Sojka, Ondřej Sojka	*The unreasonable effectiveness of pattern generation*
	9:30 am	Arthur Reutenauer	*Hyphenation patterns in TeX Live and beyond*
	10:00 am	David Fuchs	*What six orders of magnitude of space-time buys you*
	10:30 am	*break*	
	10:45 am	Tomas Rokicki, Palo Alto, CA	*Searchable PDFs with Type 3 bitmap fonts*
	11:15 am	Martin Ruckert, Hochschule Muenchen	*The design of the HINT file format*
	11:45 am	Doug McKenna, Mathemaesthetics, Inc.	*An interactive iOS math book using a new TeX interpreter library*
	12:15 pm	*lunch*	
	1:15 pm	*group photo*	
	1:30 pm	Jennifer Claudio, Sally Ha, Oak Grove H.S.	*A brief exploration of artistic elements in lettering*
	2:30 pm	William Adams, Mechanicsburg, PA	*Design into 3D: A system for customizable project designs*
	3:00 pm	*break*	
	3:15 pm	Boris Veytsman, Sch. Systems Biology, GMU	*Creating commented editions with TeX*
	3:45 pm	Behrooz Parhami, UC Santa Barbara	*Evolutionary changes in Persian and Arabic scripts to accommodate the printing press, typewriting, and computerized word processing*
	4:15 pm	Amine Anane, Montréal, QC	*Arabic typesetting using a Metafont-based dynamic font*
	4:45 pm	Takuto Asakura, National Institute of Informatics, Japan	*A TeX-oriented research topic: Synthetic analysis on mathematical expressions and natural language*
	5:15 pm	Herb Schulz, Naperville, IL	*Optional workshop: TeXShop tips & tricks*
	6:30 pm	*banquet*	Sheraton Palo Alto
Sunday August 11	8:55 am	*announcements*	
	9:00 am	Antoine Bossard, Kanagawa University	*A glance at CJK support with XeTeX and LuaTeX*
	9:30 am	Jaeyoung Choi, Saima Majeed, Ammar Ul Hassan, Geunho Jeon, Soongsil Univ.	*FreeType MF Module 2: Integration of METAFONT and TeX-oriented bitmap fonts inside FreeType*
	10:00 am	Jennifer Claudio, Emily Park, Oak Grove H.S.	*Improving Hangul to English translation*
	10:30 am	*break*	
	10:45 am	Rishikesan Nair T*, Rajagopal C.V., Radhakrishnan C.V.	*TeXFolio — a framework to typeset XML documents using TeX*
	11:15 am	Sree Harsha Ramesh, Dung Thai, Boris Veytsman*, Andrew McCallum, UMass-Amherst, (*) Chan Zuckerberg Initiative	*BibTeX-based dataset generation for training citation parsers*
	11:45 am	Didier Verna, EPITA	*Quickref: A stress test for Texinfo*
	12:15 pm	*lunch*	
	1:30 pm	Uwe Ziegenhagen	*Creating and automating exams with LaTeX & friends*
	2:00 pm	Yusuke Terada, Tokyo Educational Institute	*Construction of a digital exam grading system using TeX*
	2:30 pm	Chris Rowley*, Ulrike Fischer, LaTeX3 Project	*Accessibility in the LaTeX kernel — experiments in tagged PDF*
	3:00 pm	*break*	
	3:15pm	Ross Moore, Macquarie Univ.	*LaTeX 508 — creating accessible PDFs*
	≈3:45 pm	*end*	

Pictures from the Annual Meeting

Here are some snapshots of the TUG 2019 meeting. Credits to Jennifer Claudio, Rishikesan Nair T, and Alan Wetmore. Thank you. You can also see a ten second movie panorama of the group at:
`tug.org/tug2019/photos/group-movie.mov`

AW

Frank Mittelbach, Michael Sharpe, Steve Grathwohl

AW

Don Knuth demonstrates the organ at his home to Nelson Beebe

AW

Dennis Claudio, Boris Veytsman, Barbara Beeton

AW

Yusuke Terada, Jennifer Claudio

RT

Jim Hefferon, Rishikesan Nair T

AW

Ondřej Sojka, Petr Sojka

AW

Martin Ruckert, Doug McKenna

JC

Don Knuth examining his gift, with Chris Rowley at the banquet

AW

The whole group. Everybody smile!

Back to the roots: TUG 2019 in Palo Alto

Henri Menke

Thursday, August 8

This year's TUG meeting stood under a very special sign. TEX returned to its birthplace, Stanford University in Palo Alto, California, USA. And not only did it return to its cradle, but also its inventor would be there, none other than the great Donald E. Knuth (DEK) himself. It was a great honor to have such a very special guest at the meeting.

The evening before the meeting, there was a reception and registration at the conference site, the Sheraton Hotel. Many participants showed up and DEK was also there to personally welcome everyone.

Friday, August 9

On the morning of the first day our current president, Boris Veytsman, officially opened the conference. The first speaker to kick off the program was Erik Braun from the CTAN team to talk about goals and difficulties CTAN is facing. A notable quote from the talk is that "the Comprehensive TEX Archive Network is neither comprehensive nor an archive". However, the team wants to change that in the future together with some new web design.

Next up Arthur Reutenauer, a current maintainer of XeTEX, enlightened us about the past, present, and future state of XeTEX. As it currently seems, XeTEX might go into maintenance mode, because the original author Jonathan Kew has moved on to other projects and major contributor Khaled Hosny, who ported XeTEX from AAT to HarfBuzz, has focused his attention on LuaTEX, leaving no competent developer. With recent progress in LuaTEX and its likely adoption of HarfBuzz, the main use cases for XeTEX will be covered, making it kind of obsolete. Right now, Arthur tries to isolate XeTEX-specific code from the engine, to maybe contribute those patches to LuaTEX.

Before the morning break, Frank Mittelbach introduced us to the LaTEX "dev" format. The "dev" format is a preview of the next official release of the LaTEX format and is shipped with TEX Live for interested users to try out. The LaTEX team hopes to attract more beta testers this way, so that problems, especially regressions, can be detected before official releases, reducing the number of required hotfixes.

After the break, Frank Mittelbach continued with a second talk about UTF-8 in LaTEX. A few years ago the "Unicode revolution" was completed and almost everything is formatted in UTF-8 nowadays. Unfortunately, the old 8-bit TEX engines do not support UTF-8 natively, so the well-known `inputenc` was introduced, which knows about the "magic" multibyte sequences of UTF-8 and does the right thing by converting the characters into the LaTEX internal character representation (LICR). This technique has been integrated into the LaTEX format last year and is available to all users.

The next talk was presented by Uwe Ziegenhagen, a LaTEX and Python enthusiast from Cologne. After a short introduction to the Python programming language, he presented two ways of combining LaTEX and Python: First, how to generate LaTEX in Python using a template engine, and second, how to execute Python from within LaTEX.

My own talk was the last before lunch, and the audience had a lot to digest, despite their empty stomachs. I talked about how to parse complex data formats in LuaTEX using the integrated LPEG library. The LPEG library provides a domain-specific embedded language for parsing expression grammars (PEG) within Lua using operator overloading. After an introduction to PEG, I demonstrated how to construct a JSON parser, which can be used, for example, to read document metadata from a configuration file.

After lunch, Dick Koch, the principal maintainer of MacTEX, told us about how Apple is continuing the war against its developers. If a macOS application is not digitally signed by its developer, the system will display a warning that the application is not trusted, which is a good thing. However, with the upcoming macOS Catalina, Apple will allow only "notarized" applications to be run, which requires the developer to apply to Apple directly for notarization. Dick outlined which changes were necessary in MacTEX to pass the notarization test, so macOS TEX users will not have to worry.

This was followed by a talk by Nate Stemen, a software developer at Overleaf, about their product. Overleaf is an online LaTEX editor which allows multiple authors to edit the same document at the same time. It is useful for beginners as well, because it saves the user the installation of a full TEX distribution on their own machine and it comes with a large pool of example documents. With over four million users world-wide, Overleaf is an expanding business. The company is also very interested in symbiosis with TUG and the TEX community.

The next speaker was Rishi T from STM DOCS in India, presenting Neptune, which is part of STM's proofing framework TEXFolio (`neptune.texfolio.org`). Neptune allows journal editors to send proofs to authors with specific queries for them to address. The talk was mostly an interactive demonstration.

Before the afternoon break, Pavneet Arora spoke about distraction-free writing with Vim and how he leveraged those features to finish his latest novel.

The last session of the day started with a talk about Knuth's book *TeX: The Program.* Because TeX is a literate program, the source code and the documentation can be generated from the same file and it should be possible to read the code like a book. However, the structure of the TeX program is bottom-up, i.e., numerous low-level details are discussed before moving to the big picture, which makes the book hard to read. The speaker, Shreevatsa R, has collected resources to aid with reading at `shreevatsa.net/tex`.

Next up, Petr Sojka reviewed the history and current usage of TeX at his home institution, the Masaryk University in the Czech Republic where the pdfTeX engine was born.

This was followed by Shakthi Kannan, who introduced his free software framework "The XeTeX Book Template" to publish multilingual books using XeTeX, based on Emacs Org-mode and TeX. He presented prominent features and shared experience in creating and publishing books using the framework.

The last speaker of the day was Jim Hefferon, who spoke about his experience helping new LaTeX users (or as he called them "the great unwashed") in the `r/latex` forum on Reddit. He presented a survey of which questions are frequently asked, and pointed out that new users tend to frequent social sites on the web much more than the TUG site.

After the sessions had ended it was time for the TUG Annual General Meeting. The main theme was how to raise money for TUG by interacting first and foremost with the institutional members. Notes from Jennifer Claudio follow this report.

Saturday, August 10

On the second day of the conference we were visited by our special guest DEK, who attended all the sessions and the banquet afterwards.

The first session of the second day was started off by Petr Sojka and his son Ondřej, who spoke about their recent effort in generating better hyphenation patterns for the Czech language.

After that Arthur Reutenauer gave another, more historical talk where he reviewed the history of hyphenation patterns in TeX. The hyphenation algorithm by Liang and Knuth has since made its way into many other programs, such as OpenOffice. Therefore it seemed appropriate to unify the mess of hyphenation patterns and make them publicly available outside of CTAN. The project is ongoing and very successful (`hyphenation.org`).

The next talk was an earthshaking announcement by DEK's former PhD student David Fuchs. He has taken TeX and METAFONT and combined them into a single program with graphical output of the generated DVI. When TeX tries to load fonts, these are created ad-hoc by METAFONT and cached. At each page, the state of the TeX engine is recorded and the difference to the previous state is cached. When then editing the text, TeX is able to preview the changes in real time, even for large documents such as *The TeXbook*.

After the morning break we heard the presentation by Tom Rokicki, maintainer of the `dvips` program. Even though `dvips` has been mostly stable for a long time, Tom was unsatisfied with the fact that when bitmapped Type 3 fonts were used, it was not possible to copy and paste text from the output PDF. To this end he implemented a new font encoding routine, which reads optional encoding data to map the font correctly. This change will be available in an upcoming version of `dvips`.

In the next presentation, Martin Ruckert introduced his newest creation, the HINT file format. The HINT file format, produced using the HiTeX program, is very similar to the output of `\showlists`, as it captures most of the important information. This information can then be used by a viewer to generate screen output which is very similar (or even equivalent) to TeX, without implementing all of the algorithms. This allows reflowing the text, resulting in a great application for eBook readers, which so far have notoriously bad typesetting quality.

The last talk before the lunch break was given by Doug McKenna about his implementation of a TeX interpreter as a library, JSBox. This library is bundled with iOS applications to typeset interactive eBooks ad-hoc. This was demonstrated using his latest book *Hilbert Curves*.

Before the program resumed after lunch it was time for the group photo. After that Jennifer Claudio entertained us with different shapes of the letter 'E' that she had seen on her way to the conference.

This was followed by a talk by Federico Garcia-De Castro, a professional composer and typesetting enthusiast, who has designed an algorithm for typesetting so-called slurs in sheet music with METAFONT. It was amazing to see how much attention he spent to detail and how superior his approach is compared to commercial scoring software.

Afterwards William Adams presented a fun little project in which he manufactured a small wooden box with a CNC machine at home. The blueprints for this box were prepared with TeX.

The last session of the day was opened by Boris Veytsman, who talked about another method to prepare commented editions of a text. In this case the target was a mathematics textbook in which the teacher's version contains extra comments all around the page. This was achieved by means of his new package `commedit`.

The next presentation was a history lesson by Behrooz Parhami about the evolution of Persian and Arabic scripts from the early days of handwritten script, over the introduction of movable type, and eventually typewriters, to computer typesetting and the problems with bitmapped fonts due to small features in the glyphs.

This talk set the stage perfectly for the next, by Amine Anane, who introduced his software "Visual METAFONT", which can trace the outlines of scanned glyphs and turn them into a variable font. The newly introduced extensibility of glyphs should eventually be respected by TEX when breaking lines to offer superior typesetting for Arabic script.

The final talk of the day was presented by Takuto Asakura. The use of mathematical markup is very heterogeneous and does not necessary reflect the corresponding semantics in scientific documents. To this end, the speaker designed a synthetic analysis which harnesses the written descriptions of formulae in natural language to assign meaning to the markup.

After the official program, Herbert Schulz ran a workshop on the macOS (LA)TEX editor TeXShop.

In the evening of that day a lovely and delicious banquet took place at the Sheraton hotel. We were honored by the presence of our special guest DEK. Dinner was followed by a raffle of TAOCP set of physical books and two e-book vouchers, all donated by Pearson, as well as two TEX lion plushies donated by Jill Knuth, and a framed original black and white conference drawing. For her efforts on the local organizing committee, Boris presented Jennifer Claudio with this year's Duane Bibby signed original color conference drawing. In addition, Cheryl Ponchin and Sue DeMeritt's long service to TEX and TEX Users Group was recognized with a personalized gift certificate, as they retired from the TEX Users Group board this year. Barbara Beeton, a charter member of TEX Users Group and the TEX Users Group board, among many other TEX and TEX Users Group activities, was also recognized, with the first lifetime membership to TEX Users Group, on the occasion of her retirement from the AMS. Barbara was also given a personalized gift certificate and other group memorabilia. Finally, Don Knuth was given unusual books of organ music as a small token of appreciation, on behalf of the entire TEX community.

Sunday, August 11

The last day of the conference was begun by Antoine Bossard, who teaches at Kanagawa University in Japan. There he is confronted with typesetting mixed CJK and Latin content, so he presented his minimal approach for TEX macros to facilitate this.

The next talk was delivered by Jaeyoung Choi, who in collaboration with others designed a module for the FreeType library to render METAFONT-generated and TEX-oriented bitmap fonts. FreeType is used on many platforms including Android and Apple operating systems.

In the last talk of the first morning session, Jennifer Claudio reported on a project she undertook with her student Emily Park. They studied whether machine learning techniques could be used to detect transliterated English words in Korean text with the Hangul alphabet. As of now it seems that measures such as grayness are not sufficient to distinguish.

The second sessions started with Rishi T from STM DOCS in India, with the second presentation on STM's proofing framework TEXFolio, which is a complete journal production system that supports LATEX and XML input and HTML5 and ePub output.

Then we moved on to the next talk by Boris Veytsman. He had applied machine learning techniques to BIBTEX datasets. Starting from an annotated set of BIBTEX records he had collected from online sources, a neural network was trained to identify author, title, journal, etc., from the generated output. So far the results are mixed because citation styles vary in a rather inconvenient fashion. Most tend to abbreviate authors with initials, and physics journals often omit titles.

Before lunch we heard about another TEX format that does not come up very often but is nevertheless very important — Texinfo. It is a format for software documentation that can produce a number of different outputs including HTML and PDF. The speaker Didier Verna is applying Texinfo to the Common Lisp ecosystem to automatically generate documentation for all of the available libraries (numbered in the thousands): `quickref.common-lisp.net`.

After lunch Uwe Ziegenhagen presented his second talk, this time on creating and automating exams with LATEX using the `exam` package. Again using Python, he created different versions of the same exam to make it harder for students to copy.

This was followed by another talk on exams, by Yusuke Terada, who wants to optimize marking of the Japanese national exam called the "center test". To this end he created machine-readable exam sheets using TEX and matching software which presents the extracted answers to an examiner in an anonymized fashion to remove bias. The marks are collected electronically and reunited with the personalized information to generate an evaluation sheet. So far this system has only been implemented at Yusuke Terada's school, but should eventually become the national standard.

The conference concluded with two talks on accessibility, the first of which was delivered by Chris Rowley of the LaTeX3 team. He reported on the current state of accessibility in LaTeX and introduced the `tagpdf` package by Ulrike Fischer which aids in tagging LaTeX-generated PDFs with the proper structural elements. There are still a lot of open problems, especially concerning mathematics.

After that Ross Moore, who joined via video from Australia, demonstrated that accessibility is already possible in LaTeX if one is aware of certain difficulties, using the example of a research report he prepared for the U.S. National Park Service.

Conclusion

In summary, the TUG 2019 meeting in Palo Alto was great. Many topics were touched on and it was amazing to see which recent developments are taking place. There were a lot of lively discussions, especially with participants from the big Silicon Valley companies and it was a great honor to meet DEK. Next year's TUG 2020 will take place at the Rochester Institute of Technology in Rochester, New York.

Acknowledgment

I'd like to thank the TUG bursary for funding, which supported me in attending this conference.

◇ Henri Menke
Dunedin, New Zealand
`henrimenke (at) gmail dot com`

TUG 2019 Annual General Meeting notes

Notes recorded by Jennifer Claudio

The TUG Annual General Meeting took place during the TUG 2019 conference in Palo Alto, California, on 10 August 2019. The meeting was conducted by the TUG president, Boris Veytsman.

Boris opened the discussion by reporting the financial state of TUG and posing a question about what we can do to improve it.

An attendee asked the question regarding renewals issue, mentioning that DANTE has an automatic renewal system that TUG does not have, and perhaps a larger size to call out the renewal vs. early bird would be helpful. Henri pointed out the issue that he went for the early bird renewal but didn't renew after. Another attendee commented that since the Board is represented by more of an American base, there is a lower likelihood of having an automatic renewal system due to regional payment laws.

An attendee raised the question about options to have monthly *TUGboat* online electronically to save printing costs, and whether that actually provided savings. A suggestion was to have a trial membership that could be electronic membership only, hence a person would need a full membership in order to receive more benefits. In relation to this conversation, Alan Wetmore questioned how many beginners would be after *TUGboat*.

There was discussion that the physical copy of the *TUGboat* itself goes beyond just the user group; it is one of the few places where people can do research level publication in document processing. Frank Mittelbach noted that it is a library resource and pointed out that the ACM digital library has no physical form, which has been detrimental to it.

Frank also noted that we are not getting (as user group members) the "great unwashed" that Jim Hefferon alluded to in his talk.

It was reported that TUG membership is declining but TeX usership is not necessarily doing the same. Frank pointed out that people are getting information for free as opposed to getting benefits from community membership, and consequently proposed that institutional members should pay more to help cover the costs of the individual users and developers. This led to a following discussion that the membership needs to provide an advantage to its members. William expressed a desire to see the feasibility of a donation method. Attendees agreed that the method should not be the banner approach of sites such as Wikipedia, but should serve a similar purpose.

It was reported that 8000 people are using the LaTeX project website, of which a large proportion are actually newcomers.

This raised the question of reliance on donations from corporate/institutional users, and if that is the case, what methods should be used to increase donations. An attendee suggested it would be difficult and would rely on collaboration with sites such StackExchange and Overleaf.

Discussion ensued that the community is moving into more of a cloud-based environment, but that is not a particular goal of the user group. From the user group perspective, developers feel like they are producing the front end of something that is being used in commercial ways (e.g., Overleaf, etc.).

The question was raised as to whether TUG would be able to buy an advertisement on StackExchange. A response was that it could be considered since a community ad currently exists in a sidebar. Another person asked if it would be possible to have a hotlink pointing to a donation page.

Chris Jimenez, as a new person, expressed that he realized that the user group is an important component. He noted that he sees the efforts of the developers and that LaTeX has more visibility. He noted, however, that there is not a lot of incentive to join the group itself. People tend to gravitate toward the easiest or most convenient solutions, rather than seeking the group. He is a Word user who has found he needs more than what Word and InDesign offer, hence is new to TeX.

Jim Hefferon asked a marketing question: Is it possible on StackExchange to have a flare that says "I'm a DANTE member" or "I'm a TUG member" to show where the cohort giving answers is coming from, in order to help make the user groups known.

Federico Garcia-DeCastro expressed that he has a love and hobby for TeX, which is not what would make him pay for membership, but after attending the meeting, he feels connected. The TeX Live DVD or such isn't what makes him want to join the group.

Cheryl Ponchin asked if the group thought it would be possible for representatives at universities, and possibly at high schools, to post print media to entice users to join the user group. A high school or undergraduate initiative could include a poster competition or hosting a high school poster session.

Federico pointed out that this kind of marketing brings potentially TeX and LaTeX users, but does not bring in members to the user group.

Chris Rowley mentioned we have plenty of links with people, including founders.

Discussion returned to how many library memberships exist, since those would confer a huge potential for using *TUGboat* for library subscriptions to draw in funding. Robin clarified that subscriptions must be kept separate from membership subscriptions in order to engage university faculty. The question was raised as to whether marketing campaigns should be directed to librarians.

Robin suggested that perhaps TUG needs to redefine the TeX Users Group. She said that TUG cannot compete with Overleaf and its 4.5 million users or other commercial enterprises. She said that years ago a newbie attending a TUG conference relayed in his talk the highly unusual fact that when users write to (LA)TeX-related support lists they are essentially getting answers from the top: developers, professionals, the elite, the people who wrote the code. Robin suggested that TUG should perhaps stop wasting its time on nickel and dime issues, and focus on old and new contributors and developers who keep the language of TeX alive, relevant and flourishing. Some form of TeX is used by Overleaf, Adobe, Wikipedia, MathType, and many others, and perhaps there could be dialogue (as Boris talked about) between TUG and commercial enterprises to offer grant money or a bonus or some form of advertising, credits, something! to acknowledge the work of the TeX community. She emphasized that the large commercial enterprises should somehow support and reciprocate the generosity of the developers of the TeX Live software and related products (such as the accessibility effort). She suggested TUG help find additional funding for major projects that developers could apply for. The core membership and donations continues to support the office, overhead and committed funds such as the bursary and smaller TeX development projects, but perhaps larger grants, other funding sources, could be found for development projects with the help of seed money from commercial sources. This could include a slice for TUG, thus a well-deserved infusion of capital for conferences, bursary, etc.

Didier Verna mentioned his experience with the LISP community: back in the days there was the Association of Lisp Users because it was a young language and people with common interests needed federation, but as soon as it was standardized, the organization essentially vanished because the tool was standardized to a high degree and people were using it broadly. The end user had no incentive to join a user group because all resources had become available.

There was a suggestion that maybe the reality of the technical fields is that there is a saturation point where people know about and/or use TeX. Emails are not coming from the tech fields, but rather from the humanities.

Another question that was raised was whether PDF usage decreased relative to the use of webpages that have built-in PDF readers?

Adobe: licensing fees do not fly with most companies because they feel like they are held hostage. A better approach would be the idea of consultation and fees that could come back to TUG. It would save time for a company such as Adobe if they were able to get fast feedback or support from TUG.

Didier suggested having a fundraiser, but Boris pointed out that we don't have extra money to make a fundraiser happen. Seeking grants, as mentioned above, was one option that was raised. ◇

What do today's newcomers want?

Jim Hefferon

Abstract

Social media gives us a chance to hear directly from today's newcomers about what they are working on and what hurdles they have.

1 Introduction

Helping users is a goal of the TeX Users Group. So insight into what today's beginners need is potentially useful. Here we shall argue that social media gives us a chance to listen in on beginners, and that Reddit is a good place to do that. Then we shall present some statistics from that site about what these newcomers say.

Social media has many aspects that are like the posting boards that people in the TeX and LaTeX community have used for years. One thing that is new is that interacting in this way has become mainstream so we can expect that many people will be comfortable speaking up there. This includes people who are newcomers to our community. Some of the things they discuss are surprising.

2 Where are they?

Reddit is a news aggregation and social web site, at `http://www.reddit.com`. Members submit content to the site such as links, images, or text posts. Posts are organized by subject into user-created subreddits, which cover a variety of topics. Site members vote these up or down and submissions with more votes are displayed at the top of their subreddit. In addition, over time new posts replace older ones.

There are many subreddits, more than a hundred thousand. They are named with the prefix `r/`. One is `r/latex`, at `www.reddit.com/r/latex`, for discussions about LaTeX and TeX in general. (There is also `r/tex` but it gets very little traffic.)

The `r/latex` page looks similar to other boards that TeX and LaTeX users have seen. A typical day has a list of posts, which are usually questions, for a visitor to read and for site members to vote on. They can also comment on the post, perhaps by answering the question.

2.1 Demographics

For us, the key point about Reddit is that it is the sixth most visited website in US and twenty-first in the world, with 542 million monthly visitors (as of March 2019). The site is predominantly in English: 54% of users are from the United States, then 8% from the UK, and 6% from Canada. The `r/latex` subreddit has 19,000 members. There are a small number of posts each day and the atmosphere is relaxed and polite.

Reddit attracts young people, as this comparison with the general US shows.

	18–29	30–49	50–64	65+
US	22%	34%	25%	19%
Reddit	64%	29%	6%	1%

Users average fifteen minutes per day on the site, usually lurking.

So the first argument for the presence of newcomers here is simply the clustering of a good number of people of the right age.

2.2 Architecture

The second argument is about the alternatives. For many *TUGboat* readers the first board that comes to mind is Stack Exchange, `tex.stackexchange.com`. Here too, the page contains a list of posts consisting of questions on which site members can vote, answer, or comment.

But the culture is very different. The *About* page says, “The goal … is to be an exhaustive and curated library of detailed answers to every question related to TeX.” It strives to be all business, “This site is all about getting answers. … There's no chit-chat” and, “Questions that need improvement may be closed until someone fixes them, or just closed.” The success of the site shows that this social engineering is very effective, indeed.

However for newcomers this can be discouraging. Being told that your query is closed can be off-putting, even though you are told this politely. Beginners may feel that they don't know enough to be able to state a precise question or to search for one in the past that is related to theirs. And, a person who lurks will see lots of stuff that is far beyond them.

So, an 18–29 year old new LaTeX-er who occasionally scans the contents of `r/latex` may find more of interest, and perhaps a more amenable atmosphere, than at Stack Exchange.

A word about two additional familiar English-language boards. The Usenet group `comp.text.tex` has been around for ages. But it doesn't attract newcomers because it has become mostly a CTAN announcement list. Another long time board, and a great resource, is `texhax`. But it is not as well-known as the others to newcomers and it is low traffic so it would not reward lurking.

Thus, a second reason that `r/latex` has a disproportionate number of beginners is that the history or engineering of other places may nudge those beginners over.

3 Results

I have been a regular on `r/latex` for more than a decade. Some of the things I have found there are surprising. To quantify them I collected some data.

3.1 Data

I grabbed one thousand posts, covering `r/latex` from 2018-Nov-10 through 2019-July-08, and characterized each post in a few ways. There is a good deal of judgment involved in these characterizations but despite this noise, the numbers tell an interesting story.

3.2 Findings

First, many people make clear that they are beginners, often simply by saying it. I counted 265 authors as beginners, 29 as experienced, and 703 posts were not clear. (The numbers do not add to 1000 because of some spam.) That is, many posts begin like, "I'm a total noob ...," supporting the earlier analysis that this site attracts this group.

I suspected that many of today's beginners do not install TeX on their computer but rather start at an online site such as *Overleaf* or *CoCalc*, so I also characterized the posts by computing platform. As TeX and LaTeX are in many ways platform-independent, the great majority of posts, 817, did not name the platform. Of the remaining, 17 were using GNU/Linux, 22 were Macintosh, 63 were Windows, and the largest number, 78, were online.

The biggest challenge was to characterize the post's subject. In some cases there was more than one subject and I judged what was the main one. Here are the numbers; I'll expand on the keys below.

Key	*Number*
wrapper	92
biblio	81
graphics creation	78
resume	31
thesis	16
article	14
pandoc	13
book	13
presentation	12
classwork	12
unknown	169
other	466

The "wrapper" refers to editors or other creation environments, from `vim` to TeXworks to *Overleaf*. Thus, close to ten percent of all posts are from folks, often newcomers to LaTeX, who say they are struggling with an inability to do something that is, in some sense, not TeX or LaTeX.

Often beginners have trouble distinguishing the wrapper from TeX so they may ask, "how to get LaTeX to find and replace?" They are often not sure whether they are using `pdflatex`, or `xelatex`, etc., because that is hidden in a submenu. This is an example where the sophisticated systems available to beginners today can at least to some extent prevent them from understanding what they need to do to move forward.

Number two on the most-asked list is bibliographies. This is an area where we can perhaps do a better job helping people. For instance, on `r/latex` I often urge people to get started with LaTeX by reading the *Not So Short Introduction* [3] but I note that this document has less than two pages on this topic. Also perhaps of use would be a web page like the *LaTeX Font Catalogue* [2], but instead containing a selection of bibliography styles.

The next most asked topic is another one that many experienced users also have trouble with, creating graphics. This usually takes the form of, "In Ti*k*Z, how do I ... ?"

After that, the next item is a surprise, at least for me: today many people, as an early LaTeX encounter, write a resume. One factor may be that because resumes have rather complicated formatting, they could lead to a disproportionate number of posts.

Following those are subjects that many readers may have expected. This includes questions about thesis and article styles, styles for books, and questions about presentations using Beamer. It also includes using TeX and LaTeX for class notes or homework. (Pandoc is a program to convert files among markup languages.)

The "other" category is large but also scattered. Subjects that appeared include fonts, figures, and tables. Also there are links to blog posts about LaTeX topics.

3.3 Observations

I will close with a few comments comparing newcomers on `r/latex` with those in the past.

First, delightfully, missing from the board discussion are many things that in the past gave newcomers trouble. There are not many questions about installing an entire system, rarely are people stuck on the "Hello World" problem of getting that first document out the other end, and no one ever asks about tuning font parameters for a printer.

Surprising to me is that many posters introduce themselves as undergrads. No longer is the first encounter with TeX and LaTeX, and our community, restricted to professionals and graduate students.

Perhaps related to the prior point is that often posters acknowledge a sense that TeX does the best documents. So there is widespread awareness among this young group of the power of TeX and LaTeX and friends.

These beginners often ask for a "template," by which they mean a file into which they can drop their content. For these, people are often pointed to *Overleaf*.

Finally, related to that, newcomers have typically not looked on CTAN, or even heard of CTAN. And if they have gone there, they can be stymied by a paradox of choice.

3.4 Just ask them

As a follow-up to the survey, I posted: *If you are a beginner then what would help you in TeX and LaTeX? ... What do you find to be the biggest hurdle?* [1] There were about a dozen responses, which make interesting reading (see the link in the citation). These are largely in line with the description above so that respondents described problems with packages, including finding a suitable one or understanding interactions and conflicts among packages. And, they expressed struggling with Ti*k*Z.

4 Summary

There are reasons to suppose that social media can help us understand what beginners today are working on and struggling with. Analysis shows that some real stumbling blocks from the past are not present today and tells us a little about what does give newcomers trouble. It also shows that today's beginners are younger than has been traditional.

References

[1] JimH10. Are you a newcomer to LaTeX? `https://www.reddit.com/r/LaTeX/comments/ccc93c/are_you_a_newcomer_to_latex/`, 2019. [Online; accessed 2019-August-13].

[2] Palle Jørgensen. The LaTeX Font Catalogue. `https://tug.org/FontCatalogue/`, 2019.

[3] Tobias Oetiker. The Not So Short Introduction to LaTeX 2ε. `https://ctan.org/pkg/lshort`, 2019.

◇ Jim Hefferon
Saint Michael's College
`jhefferon (at) smcvt dot edu`

Type 3 fonts and PDF search in `dvips`

Tomas Rokicki

Abstract

PDF files generated from the output of `dvips` using bitmapped fonts have not been properly searchable, indexable, or accessible. While a full solution is challenging, only minimal `dvips` changes are required to support English language text, changes that are at least two decades overdue. I will describe these changes and discuss their limitations.

1 Introduction

The Type 3 fonts generated by `dvips` for bitmapped fonts lack a reasonable encoding vector, and this prevents PDF viewers from interpreting those glyphs as text. This in turn prevents text search, copy and paste, screen readers, and search engine indexing from working correctly. Fixing this is easy, at least for English text, and comes with no significant cost.

This is not nearly a full solution to create accessible multilingual PDF documents. Support for eight-bit input encodings [2], explicit font encodings [3], and direct generation of PDF can yield better results. But if you want to use METAFONT fonts as-generated and `dvips`, this is an important change.

I describe how I generated reasonable encoding vectors for common METAFONT fonts, how `dvips` finds these encoding vectors and embeds them in the PostScript file, and how the current implementation allows for future experimentation and enhancement.

2 A little history

When `dvips` was originally written in 1986, the lone PostScript interpreter on hand was an Apple LaserWriter with 170K available memory. I treated PostScript as just a form of compression for the page bitmap, doing the bare minimum to satisfy the requirements for Level 1 Type 3 fonts. One of those requirements was to supply an `/Encoding` vector, despite the fact that at the time, the vector was completely unnecessary in rendering the glyphs. Not considering that people might someday use that encoding vector for glyph identification, on that fateful day in 1986 I generated a semantically nonsensical but syntactically acceptable vector (`/A0`–`/H3` in base 36) for all bitmapped fonts, and this vector remains to this day, subverting any attempt to search copy, or use screen readers.

Replacing this encoding vector with something more reasonable allows PDF viewers to properly understand what characters are being rendered, at least for English-language text.

3 A sample

The following TEX file, cribbed from `testfont.tex` but using only a single font, will be used for illustration.

```
\hsize=3in \noindent
On November 14, 1885, Senator \& Mrs.~Leland
Stanford called together at their San
Francisco mansion the 24~prominent men who
had been chosen as the first trustees of The
Leland Stanford Junior University.
?‘But aren’t Kafka’s Schlo{\ss} and {\AE}sop’s
{\OE}uvres often na{\"\i}ve  vis-\‘a-vis the
d{\ae}monic ph{\oe}nix’s official r\^ole
in fluffy souffl\’es?
\bye
```

When you run this through TEX and `dvips` (giving the `-V1` option to enforce bitmapped and not Type 1 fonts), and then `ps2pdf`, the resulting PDF does not support text search in most PDF viewers. In Acrobat with copy and paste it almost works; the c's are dropped throughout (San Francisco becomes San Fran is o). The c's are dropped because the original `dvips` encoding uses `/CR` as the name for this character, and it is apparently interpreted as a non-marking carriage return. Ligatures also don't work. In Mac OS X Preview (the default PDF viewer for the Mac), selecting text appears to fail (it actually works, but the selection boxes are too small to see that anything has actually been selected) and no characters are recognized as alphabetic. In Chrome PDF preview, selecting text gives a random note appearance with each word separately selected by its bounding box and no alphabetic characters recognized.

Conversely, when you process the file with Type 1 fonts, all text functions perform normally, except that accented characters are detected as two separate characters (the accent and the base character). The critical difference is not Type 3 (bitmaps) versus Type 1 (outline fonts), but rather the lack of a sensible encoding vector in the Type 3 font.

4 First attempts and failure

If I manually copy the `Encoding` vector from the output of dvips using Type 1 fonts and put that in the font definition for the Type 3 fonts, the situation improves; now Adobe Acrobat properly supports text functions (including ligatures but not accented characters). The other PDF viewers now recognize alphabetic characters, but they still have a number of problems.

With Preview, if you use command-A (to select all the text) and then command-C (to copy it), and then copy the result into a text editor (or a word processing program "without formatting"), you get the following mishmash of text:

> On Novemb er 14, 1885, Senator & Mrs. Leland Stanford called mansion the 24 together at their San Francisco prominent men who had b een cho- Stanford sen as the first trustees of The Leland Junior Æsop's University. ¿But aren't Kafka's Schloß and Œuvres often na"ıve vis-‘a-vis the dæmonic phœnix's official rˆole in fluffy souffl'es?

In addition to the broken words and split accented characters, if you look carefully you will notice some surprising and substantial word reordering! What could be going on?

5 Refinements and success

All PDF viewers use some heuristics to turn a group of rendered glyphs into a text stream. The heuristics differ significantly from viewer to viewer. The most important heuristic appears to be interpreting horizontal escapement into one of three categories: kerns, word breaks, and column gutters. Preview was failing so badly because it was recognizing rivers in the paragraph as separating columns of text. To satisfy the PDF viewers I had access to, I made two additional modifications to each bitmapped font.

First, I adjusted the font coordinate system, as defined by the so-called font matrix. The default Adobe font coordinate system has 1000 units to the em, while the original `dvips` uses a coordinate system with one unit to the pixel both for the page and for the font, and doesn't use the PostScript `scalefont` primitive. But not using `scalefont` apparently makes some viewers think all the fonts are just one point high, and they use spacing heuristics appropriate for such a font. By providing a font matrix more in line with conventional fonts, and using `scalefont`, PDF viewers make better guesses about the appropriate font metrics for their heuristics.

Second, I provide a real font bounding box. The original `dvips` code gives all zeros for the font bounding box, which is specifically allowed by PostScript, but this confuses some PDF viewers. So I wrote code to calculate the actual bounding box for the font from the glyph definitions.

With these adjustments, using `dvips` with bitmapped fonts and `ps2pdf` generates PDF files that can be properly searched with most PDF viewers — at least, for English language text.

6 Other languages: No success

I would have liked things to work with other languages as well, but was not able to get it to work. Clearly the PDF viewers are recognizing characters by the glyph names, but this appears to work only with a small set of glyph names. I hoped that those listed in the official Adobe Glyph List [1] would work, but in my experiments they (for the most part) did not. I also tried Unicode code point glyph names such as `/uni1234` and `/u1234` but neither of these formats worked in the PDF viewers I tried. I also experimented with adding a `cmap` to the font, with no success, and even tried some lightly documented GhostView hacks, but was able to achieve only distressingly partial success for most non-Roman characters.

Even if the individual glyphs are recognized, problems remain with accents, and more generally, virtual fonts. With a standard seven-bit encoding, accents are generally rendered as two separate characters, where the PDF viewer expects to see only a single composite character. Further, the entire virtual font layer would need to be mapped in some fashion, as the PDF contains the physical glyphs that are often combined in some way to provide the semantic characters. Supporting this would have required significantly more effort and heuristics, and there are already efforts in this direction from people much more knowledgeable and capable than I am. The most logical general solution is to use properly coded input, such as UTF-8, and where transformation to multiple glyphs is necessary, embed the appropriate mapping information directly in the PDF file.

The lack of success for other languages diminishes these proposed changes, but the changes are still important as they do provide reasonable support for English-language documents. Since PDF viewers are a moving target, as are the PostScript to PDF converters, the implementation provides for some future experimentation and extension.

7 Finding font encodings

In order to provide more than a proof of concept, I had to determine appropriate glyph names for the fonts provided with TEX Live, as well as provide a mechanism for end users to add their own glyph names for their own personal fonts.

Over the years others have translated nearly all of the METAFONT fonts provided with TEX Live, and as part of that process, reasonable encoding vectors have been created for the glyphs. I decided to leverage this work, so I wrote a script that located all the METAFONT sources in the TEX Live distribution, all the corresponding Type 1 fonts, and any encoding files used in the relevant `psfonts.map` file. A big Perl script chewed on all of this, extracting encoding vectors and creating appropriate files for `dvips`. Some of the encoding vectors use glyph names that are not particularly useful, and some use glyph names based on Unicode code points that are not currently recognized by the PDF viewers I tried. I did not want to edit the names in any way; I aimed for functional equivalence to using the Type 1 fonts. If improvements are made to the Type 1 font glyph names, or to the PDF viewers, I wanted to be able to pick up those improvements.

I considered having `dvips` read the encoding vectors directly from the Type 1 fonts, rather than extracting them and storing them elsewhere, but decided against this; I wanted `dvips` to use appropriate glyph names even if the Type 1 fonts didn't exist at all. This does introduce redundancy which can potentially lead to an inconsistency in the glyph names, but the fonts are currently mostly stable, and the glyph name extraction process can be repeated as needed if meaningful changes are made.

8 Storing and distributing encodings

After scanning all of the relevant METAFONT files and corresponding Type 1 files, I found there were 2885 fonts; storing the encodings separately one per font would require an additional 2,885 files in TEX Live, occupying about 5 megabytes. I felt this was excessive for the functionality added.

Karl Berry suggested combining all the encodings into a single file, along with a list of fonts using any particular encoding. Since there were only 138 distinct encodings, this gave tremendous compression, letting me store all of the encodings for all of the fonts in a single file of size 183K. This also enabled me to distribute a simple test Perl script that mimicked the changes so people could try them out without updating their TEX installation.

This combined file, called `dvips-all.enc`, provides the default encoding used by the 2885 distributed TEX Live METAFONT fonts. In every case that `dvips` looks for an encoding, e.g., for `cmr10`, it first searches for `dvips-cmr10.enc` and only falls back to the information in the combined file if the font-specific file is not found. This permits users to override the provided encodings, as well as define their own encoding for local METAFONT fonts.

The format of the encoding file is slightly different from that of other encoding files in TEX Live. The encoding file should be a PostScript fragment that pushes a single object on the operand stack. That object should either be a legitimate encoding vector consisting of an array of 256 PostScript names, or it should be a procedure that pushes such an encoding vector. It should not attempt to define the `/Encoding` name in the current dictionary, as some other encoding file formats do. A sample file, one that can be used for `cmr10` (and many other

Computer Modern fonts) is:

```
[/Gamma/Delta/Theta/Lambda/Xi/Pi/Sigma/Upsilon
/Phi/Psi/Omega/ff/fi/fl/ffi/ffl/dotlessi
/dotlessj/grave/acute/caron/breve/macron/ring
/cedilla/germandbls/ae/oe/oslash/AE/OE/Oslash
/suppress/exclam/quotedblright/numbersign
/dollar/percent/ampersand/quoteright/parenleft
/parenright/asterisk/plus/comma/hyphen/period
/slash/zero/one/two/three/four/five/six/seven
/eight/nine/colon/semicolon/exclamdown/equal
/questiondown/question/at/A/B/C/D/E/F/G/H/I/J
/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z/bracketleft
/quotedblleft/bracketright/circumflex
/dotaccent/quoteleft/a/b/c/d/e/f/g/h/i/j/k/l
/m/n/o/p/q/r/s/t/u/v/w/x/y/z/endash/emdash
/hungarumlaut/tilde/dieresis
128{/.notdef}repeat]
```

9 Deduplicating encodings

The encodings inserted in the fonts do use a certain amount of PostScript memory, and this memory usage is not presently accounted for in the memory usage calculation of `dvips`. The memory usage is small and modern PostScript interpreters have significant memory. Further, I doubt anyone actually sets the `dvips` memory parameters anymore anyway. So this is unlikely to be an issue. But to minimize the effect, and also to minimize the impact on file size, encodings that are used more than once are combined into a single instance and reused for subsequent fonts.

10 The `dvips` Changes

Almost all changes to `dvips` are located in the single new file `bitmapenc.c`, although a tiny bit of code was added to `download.c` to calculate an aggregate font bounding box, and the font description structure extended to store this information. I also added code to parse command line options and configuration file options to disable or change the behavior of the new bitmap encoding feature.

By default this feature is turned on. If no encoding for a bitmapped font is found, no change is made to the generated output for that font.

11 Testing the changes without updating

You can test my proposed changes to the `dvips` output files without updating your distribution or building a new version of `dvips`. The Perl script `addencodings.pl` [4] reads a PostScript file generated by `dvips` on standard input and writes the PostScript file that would be generated by a modified `dvips` on standard output. No additional files are required for this testing; the default encodings for the standard TeX Live fonts are built into the Perl script.

12 How to use a modified `dvips`

In general, `dvips` usage is unchanged. Warnings in the functionality of the bitmap encoding are disabled by default, so as to not disturb existing workflows; this may change in the future.

I add a single command line and configuration option, using the previously unused option character `J`. The option `-J0` disables the new bitmap encoding functionality. The option `-J` or `-J1` enables it but without warnings, and is the default. The option `-J2` enables it with warnings for missing encoding files.

13 Extension support

Remember that the encoding file is an arbitrary PostScript fragment that pushes a single object on the operand stack, and that object can be a procedure. I permit it to be a procedure to support experimenting with other changes to the font dictionary to improve text support in PDF viewers. For instance, if a technique for introducing Unicode code points for glyphs into a PostScript font dictionary is found and supported by various PostScript to PDF converters, such a procedure could introduce the requisite structures. The procedure will not be executed until the font dictionary for the Type 3 font is created and open.

To test this functionality, I created a `rot13.enc` file that defines a procedure that modifies the Encoding vector to swap single alphabetic characters much like the `rot13` obfuscation common during the Usenet days. With this modification, copying text from a PDF copies (mostly) content that has been obfuscated (except for ligatures). This brings us full circle to the current unreadable text copied from the original `dvips`.

References

[1] Adobe. Adobe glyph list specification. `https://github.com/adobe-type-tools/agl-specification`, August 2018.

[2] A. Jeffrey and F. Mittelbach. `inputenc.sty`. `https://ctan.org/pkg/inputenc`, 2018.

[3] R. Moore. Include CMap resources in PDF files from pdfTeX. `https://ctan.org/pkg/mmap`, 2008.

[4] T. G. Rokicki. Type 3 search code. `https://github.com/rokicki/type3search`, July 2019.

◇ Tomas Rokicki
Palo Alto, California
United States
`rokicki (at) gmail dot com`

The state of XƎTEX

Arthur Reutenauer

Abstract

XƎTEX was the first TEX engine to support Unicode natively and was actively developed until recently, but has since then gone into maintenance mode. I will discuss avenues for future development.

0 XƎTEX & LuaTEX

Let's start with a quick comparison between XƎTEX and LuaTEX, its Unicode-supporting cousin. While both are similar in their overarching goals to support modern encodings and font standards, they differ in an essential tenet of their philosophies: XƎTEX transplants a lot of additional features into the core by means of external libraries, while LuaTEX opens up the engine by allowing large parts of it to be rewritten in the Lua scripting language (the surgical metaphor is freely borrowed from Hans Hagen, main developer of ConTEXt and designer of LuaTEX).

This is quite a significant difference. XƎTEX's architecture enables it to delegate crucial tasks, notably *shaping* (the processes necessary to display complex scripts correctly, such as Arabic and Indic). The library currently used for that task is called HarfBuzz, and was integrated by Khaled Hosny in 2012–2013. Conversely, LuaTEX depends only on Lua code for the same tasks, but such code has to be written, and the only person currently doing so is Hans. This means that the number of scripts supported in LuaTEX will necessarily be limited.

On a more technical level, the core of XƎTEX still uses the original WEB code, while LuaTEX has been rewritten in C.

1 XƎTEX + LuaTEX

One idea to shake up XƎTEX was thus to use the code base of LuaTEX to progressively replace the `WEB` functions of the XƎTEX source by their C equivalent. This would be a somewhat sounder basis for future developments. In addition, we would get Lua "for free", although the interaction with LuaTEX's callbacks probably would need to be massaged quite a bit. But the prospect of taking advantage of the very large amount of work already done on LuaTEX, its comparatively higher development pace, and the possibility of merging efforts, made it a goal worth contemplating.

I have been experimenting last winter in that direction and think this effort, that we would presumably call XƎLuaTEX, is sustainable. Nevertheless, since it also entails considerable work, I have also been exploring other options.

2 XƎ + LuaTEX+ HarfBuzz

At about the same time, Khaled was working on integrating HarfBuzz into LuaTEX, to support more scripts. This could be a possible future for XƎTEX, but it should be noted that the situation currently is a little confused, since the ongoing effort inspired the current LuaTEX maintainer, Luigi Scarso, to produce his own experimental version of LuaTEX with HarfBuzz dubbed luahbTEX. It may thus be wise to wait for the dust to settle before deciding if that can be the future for XƎTEX. And there's more!

3 XƎ + `lmtx`

Another new project is the effort by Hans, always indefatigable, to overhaul LuaTEX into a leaner engine with a different build system. This `lmtx` was announced on 1 April (but wasn't an April fool's joke) and will become the basis for the next major version of ConTEXt. The first official release will be during the 2019 ConTEXt meeting, two weeks from the time of writing, hence I thought that as long as I was contemplating possible futures for XƎTEX, I might as well have a look in some detail at the upcoming `lmtx`! HarfBuzz will not be a part of it, since ConTEXt is using the Lua shaping code, hence a similar effort as the one mentioned in the previous paragraph would be needed.

4 Why?

Why, one might ask, bother with such considerations at all? XƎTEX already exists and in spite of some misfeatures (for example in the bidirectional models), it has no serious bugs. The absence of new development obviously means that it is very stable.

However, no program keeps being maintained in the long run just by staying exactly identical (TEX90 being a lone exception). XƎTEX still has essential features that are unique in the TEX world: complex scripts is the most important one; and the intercharacter token mechanism also lacks an equivalent in LuaTEX (I'm grateful to Henri Menke for bringing the latter to my attention during the conference). If the developments outlined in section 2 do give rise to an extended LuaTEX engine with all of XƎTEX's high-level capabilities, it will be time to bridge the gap by adding all the small missing bits and pieces, and merge the two projects together (which obviously is my ultimate goal). Until such time, however, experiments are in order.

◇ Arthur Reutenauer
`arthur.reutenauer (at) normalesup dot org`

Hyphenation patterns: Licensing and stability

Arthur Reutenauer

Abstract

New thoughts on old questions: hyphenation patterns, licensing, and stability.

1 Don Knuth's question

The package hyph-utf8, started in 2008 by Mojca Miklavec and myself to collect all known hyphenation patterns for different languages, has already been the subject of two *TUGboat* articles [2, 3], and the talk I gave during TUG'19 was a summary of those. Having worked on that project for over a decade, I was nonetheless caught by surprise when Don Knuth, who took the first question, asked me how we dealt with archiving and the need to keep page breaks stable over time. I improvised a reply that I'm afraid was not very convincing, and partly missed the point. Here is the answer I wish I had given.

2 Initial answer

There is no policy on stability and backward compatibility for hyphenation patterns in TEX distributions. When Mojca and I took over the existing patterns, we found no evidence of a strategy, or even a rule of thumb, to decide how to update them, and in particular no safeguard against incompatibilities introduced by correcting errors in existing patterns. Depending on contributors' availability, there could be regular updates over a period of time (usually not exceeding a few years), small improvements at irregular intervals, or — most often — no changes at all after the initial development effort.

A practical issue arose soon after we got started on hyph-utf8, as the German patterns were being worked on very actively, in an extensive effort that was guaranteed to introduce incompatibilities. There was however no doubt that such an update would be beneficial to German-speaking users, as it addressed many earlier misses and mistakes. Because we needed a decision, we followed the sensible piece of advice by Karl Berry (who was also the only person to venture an opinion), that we simply keep the patterns as-is for TEX and pdfTEX, and only use the new patterns, as well as any later updates, for XƎTEX and LuaTEX. (The same team that updated the German patterns produced a package to optionally use the "experimental" patterns in the 8-bit engines as well.) The sentiment was that it was essential to commit to some kind of stability for a major language like German, where incompatible changes would affect more users; and for the older engines, that were already mature.

All in all, however, surprisingly little discussion has ever taken place about how to achieve stability in such an essential part of any TEX installation. The main topic of the conversations we've had has indeed been rather different . . .

3 Interrogation

The problem of licences has already been discussed in [3] in connection with other projects interested in the patterns from hyph-utf8, which often had reservations about the LPPL (LATEX Project Public License) for one reason or another. Since that licence is quite central to the TEX world, and it's been used for many pattern files, I will discuss it in the next two sections.

4 What's in a licence

The LPPL was written in order to formalise the conditions that Don put on distributing TEX — anyone may freely use the idea and even the code, but a program may not call itself TEX unless it passes the `trip.tex` torture test — and boils down to:

- Any derivative work, whenever it "identifies itself to the user . . . clearly and unambiguously identifies itself" as such [clause 6(a)].
- . . . except when made by a specific person, the *maintainer*, in which case the derivative work is considered an updated version of the original work [clause 4]. A work under the LPPL can be either *author-maintained* (only the original author can ever be maintainer), *maintained* (a new maintainer could take over in the future), or *unmaintained* [section "Maintenance of The Work"].

I explained in [3] why the latter point isn't really suitable for hyph-utf8, and forgot to mention that the former wasn't either: except with LuaTEX, patterns are dumped into the format, and thus never "identif[y] [themselves] to the user" during a normal TEX run. That clause of the LPPL is simply moot for hyphenation patterns.

In other words, even if all the patterns were under the LPPL, one could very well produce a new, completely incompatible set of patterns while respecting the letter of the licence, and users wouldn't notice from looking at their terminal or log files.

It's also striking that the only mechanism the LPPL provides to ensure stability is to put everything into the hands of an all-knowing maintainer, who gets full control over the successive versions (and I do mean *full*: the only duty of a maintainer is to publish up-to-date contact details, not even to acknowledge bug reports sent through this contact).

5 What isn't

The advantage to having one person, or a few people, designated as solely responsible for a package, is clear: distributions need to know who is entrusted with making updates, with collecting bug reports and (hopefully) fixing them, etc. Nor can too many formal duties be attached to that responsibility, as that would be unfair to the often overburdened volunteers who put their time and effort at the service of the community.

Beyond these practical considerations, however, it's not clear how maintainers are supposed to ensure stability. They could of course be the ones to gather user wishes and strike compromises, but equally they could just make decisions without consulting anyone else, and in our experience the latter is much more common than the former. It even takes a more sinister turn, as with the recurring case of one prolific package maintainer renouncing his production and dumping it on the lap of hapless volunteers, only to later claim it back and try to assert what he considers his rights, by among other means declaring his package author-maintained. (It's a real problem. I am among the people trying to deal with the situation. We don't know what to tell this person.)

Clearly, no licence is in and of itself going to help with difficult people who abuse it. It can only lay out conditions and principles that its adopters will follow.

The problem is that the LPPL does not even do that. As summarised in the previous section, it deals in great details with name and maintainer changes, but doesn't actually offer any explanation of what maintainers can do to guarantee compatibility. In fact, it doesn't even use the words "compatibility", "stability", or any related ones, except in the preamble and a paragraph near the end, both of which state without explanation that the licence helps with compatibility and stability. Readers are referred to [1] to check for themselves.

Even more than specific words, what I'm missing in the LPPL is some sense of a commitment to stability — an encouragement to package maintainers to produce equivalents to `trip.tex` for example — instead of rights without corresponding duties. (It's all the more surprising as `modguide.tex`, a precursor document, did in fact mention regression tests prominently.) This spirit of the LPPL has in my opinion contributed to a certain complacency in the TeX world, an unwarranted feeling that because of its venerable origins our community is "better at compatibility".

6 A new answer

In light of the above, I hope I can be forgiven for not having a ready answer to this essential question: how can authors ensure that their linebreaks are going to be stable in the future? At this point I need to soften my initial response (lest Don should have a heart attack!) because there is *some* policy about stability: the original `hyphen.tex` will never change and will always available as `\language0` (that is hardcoded in all our software); and there is, as mentioned, a general feeling that patterns for "major languages", whatever those are, shouldn't change too much (although big changes were made to the Spanish patterns under our watch a few years ago). Apart for that, it's pretty much free-for-all. We do of course monitor the situation and discourage authors from changes that are too extensive, but decisions are made on an as-needed basis.

A more systematic approach would require an actual discussion about what compatibility means for hyphenation patterns, and how to achieve it. Among the ideas usually mooted are a complete pattern freeze; lists of hyphenated words whose breakpoints are guaranteed not to change; and a versioning system for patterns. I can't imagine that a single strategy is going to fit every language, but there could be a multiple-tier system of pattern stability, with each language mapped to one tier.

Most important, we need decisions. It's not clear to me that Mojca and I should be the ones to make them, since we're only the implementors, but we'll be more than happy to help along the way, and to execute any vision that can be had in that area. As indeed we have, for over a decade.

References

[1] LaTeX3 Project. The LaTeX Project Public License, version 1.3c, 2008. `http://mirror.ctan.org/macros/latex/base/lppl.tex`

[2] M. Miklavec and A. Reutenauer. Putting the Cork back in the bottle — Improving Unicode support in TeX. *TUGboat* 29(3):454–457, 2008. `https://tug.org/TUGboat/tb29-3/tb93miklavec.pdf`

[3] M. Miklavec and A. Reutenauer. Hyphenation in TeX and elsewhere, past and future. *TUGboat* 37(2):209–213, 2016. `https://tug.org/TUGboat/tb37-2/tb116miklavec.pdf`

◇ Arthur Reutenauer
Storgatan 28B
753 31 Uppsala
Sweden
`arthur (at) hyphenation dot org`

MacTEX-2019, notification, and hardened runtimes

Richard Koch

Abstract

MacTEX installs everything needed to run TEX on a Macintosh, including TEX Live, Ghostscript, and four GUI applications: TeXShop, TEX Live Utility, LATEXiT, and BibDesk. In macOS 10.15, Catalina, Apple requires that install packages be notarized, and all command line and GUI applications in such a package must be signed and adopt a hardened runtime. I'll explain what this means and how it was accomplished.

MacTEX 2019

1 Recent changes

For many years, MacTEX supported macOS 10.5 (Leopard) and higher, on both PowerPC and Intel processors. Starting in 2017, we decided to limit support to those systems for which Apple still provides security updates. Consequently, we support the three latest systems; in 2019 we support Sierra, High Sierra, and Mojave (that is, 10.12 and higher). Each fall, Apple introduces a new system and we also support that. Thus MacTEX-2019 will support Catalina when that is released this fall.

Mojca Miklavec compiles Mac binaries for older systems; in 2019 she supports Snow Leopard (10.6) and higher. TEX Live contains both our binaries and Miklavec's binaries. Our web pages (`tug.org/mactex`) explain how to install TEX Live using either the MacTEX installer or the standard Unix install script (`install-tl`), so users with older systems can update using the Unix install script. Both methods produce exactly the same TEX Live in the end.

2 Security

I retired from the University of Oregon in 2002. In that year, freshmen arriving at the University discovered a CD and instruction sheet taped over the ethernet jacks in their dorm rooms. The sheet said

```
Warning: You must install the virus checker on
this CD before connecting your computer to the
ethernet. If you fail to follow this instruction,
you will lose ethernet privileges in this room.
```

The note ended with one more sentence:

```
Macintosh users can ignore this message.
```

But that was 2002. This April, I got the following:

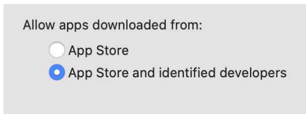
```
From: koch@math.uoregon.edu
Date: April 4, 2019
To: koch@math.uoregon.edu

Hey! I compromised your account and gained full
access to it. I just sent this email from your
account. You visited an adult website and got
infected. This gave me access to all of your
contacts, browsing history, your passwords,
your webcam, and even your microphone.

I noticed you were trying to please yourself by
watching one of those nasty videos, well my son,
I recorded your actions ... (thanks to your
webcam) and even recorded your screen (the video
you were watching). Now, if you do nothing, then
I will send this video to all of your email,
social media and messenger contacts. You have
the option to prevent me from doing all of this.
All you need to do is to make the transfer of
\$958 to my bitcoin address ...
```

3 Lessons

- The Macintosh is built on top of Unix. Unix has strong protection against *other irresponsible* users. Like most companies, Apple has security engineers patching kernel and system bugs as they are found.
- But Macs are generally used by one person, and the remaining problem is to protect that person against himself or herself. If my Mac is attacked, I'm not worried that the criminal will become root. I'm worried that he will activate my camera, read my mail, find my contact list, or turn on my microphone.
- For several years, Apple has provided a (mandatory) solution for applications in the App Store. It is known as *sandboxing*. A sandboxed application cannot interact with other programs; it runs in its own sandbox.
- In Catalina (and also to some extent in Mojave) Apple provides a different kind of security protection for other programs. Unlike sandboxing, the new security is carefully tuned to allow any program to run as usual. Here's how it works.

4 Signing

This step was introduced in 2012. Apple Developers can *sign* their applications and their install packages. When software is downloaded from the Internet, the system checks that the software has not been modified since it was signed, and that the signature is from a known developer. It refuses to run software that doesn't pass. Otherwise it sets a Finder bit to disable future checks and runs the software. A control panel in Apple's System Preferences controls this behavior:



Signing requires developer status from Apple, which costs $100 a year. TeXShop and MacTEX have always been signed.

Apple issues *two* developer signing certificates, one for applications and one for install packages. Signing applications is done in XCode as part of the build process. A command line binary signs install packages.

Tricks explained on the Internet allow users to disable the signing requirement and install any program. At this year's WWDC, Apple said that such tricks would *always* be available.

5 Notarization

This spring, Apple added notarization. This works like signing; both applications and install packages can be notarized. Once software is signed and just before release, it is sent to Apple. There it is checked for viruses (no human hands touch the software). Checking takes around 15 minutes. If the software passes the test, a "certificate" is mailed back and "stapled" to the software. In Catalina, software downloaded from the Internet must be both signed and notarized before it can run.

Previously, software was only tested once to make sure it was not modified. Now these tests will be rerun periodically. The details are somewhat vague (to me), so don't ask.

6 Hardened runtimes

Signing and notarization are small potatoes. The big security step in Catalina is the requirement that all applications and command line programs in a notarized install package must be signed and timestamped, and must adopt a Hardened Runtime. All of this is new. The MacTEX install package has been signed since 2012, but the individual TEX binaries are not signed. And while TeXShop is signed, the remaining applications TEX Live Utility, LATEXiT, and BibDesk are not signed. The kicker, however, is that these applications *and all command line apps* must adopt a hardened runtime. What is that?

Apple has a list of 13 dangerous operations a program might try to perform. I'll give the full list later, but among the items are these: accessing the camera, accessing the microphone, accessing location information, accessing the address book, accessing the user's calendars, accessing photos, sending Apple events to other applications, executing JIT-compiled code, loading third party libraries not by Apple. If an application adopts a hardened runtime, it is not allowed to perform any of these operations.

However, for each of the 13 dangerous operations, a developer can claim an *entitlement*. I have always dreamed of a TEX editor attached to a camera; to make a commutative diagram, draw it and take a picture and the editor converts the drawing into TEX. The author of such an editor would file an entitlement for the camera operation.

Nobody at Apple checks the entitlement list; there is no "approval process". A developer can claim all 13 entitlements and then the hardened runtime has no effect.

So calm down that case of paranoia. Apple isn't restricting developers. It is providing a tool to help open source developers improve security.

6.1 Dealing with command line programs

Command line programs can adopt a hardened runtime without recompiling. The command below does this for the `xz` binary used by `tlmgr`. The `--force` option says to replace any previous signing by the new one, and `--options=runtime` says to adopt a hardened runtime with no exceptions.

```
codesign \
 -s "Developer ID Application: Richard Koch" \
  --force --timestamp  --options=runtime xz
```

To claim exceptions for a command line program, add a flag `--entitlements=TUG.entitlement` to the previous call, where `TUG.entitlement` can be any name and is a short XML file. The example `TUG.entitlement` here allows linking with third party libraries. (One long line has been broken for *TUGboat* with a \; it should not be broken in a real file.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
 "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.security.cs.\
disable-library-validation</key>
  <true/>
</dict>
</plist>
```

By embedding the `codesign` call in a shell script, it is easy to construct scripts which sign, timestamp, and adopt hardened runtimes for all command line binaries in an install package.

6.2 Case 1: BasicTEX

In addition to the full MacTEX, we provide a smaller install package called BasicTeX, which installs the distribution obtained by using `install-tl` with the "small" scheme. To test the above ideas, I submitted this package unmodified to Apple for notarization. Apple refused to notarize it, but they sent back a detailed and easy-to-read error sheet. The bin directory of BasicTEX has 88 items. Apple ignored symbolic links, scripts, and other files, but had problems with 30 commands. These were exactly the commands which the Unix command `file` listed as "Mach-O 64-bit executable x86_64".

In addition, Apple found three other such binaries in `tlpkg/installer`: `lz4`, `wget`, `xz`.

I used the codesign script on these 33 binaries and submitted BasicTEX again to Apple for notarization. Approved!

6.3 Case 2: Ghostscript

Ghostscript only has two binaries, `gs-X11` with X11 support and `gs-noX11` without X. We install a symbolic link named `gs` to the appropriate binary.

I ran `codesign` on `gs-X11` and `gs-noX11` and submitted to Apple. Apple notarized the install package. But when the package was used to install Ghostscript, `gs` refused to run. Why?

Originally, Apple supplied an optional install package for X11. But their package was often out of date, so a mutual decision was made for a third party to supply X11 for the Macintosh as open source. Consequently, `gs-X11` links in a third party library, which is not allowed for hardened runtimes. Resigning `gs-X11` and claiming an entitlement for such linking solved the problem.

6.4 Case 3: `biber`

The `biber` binary is so complicated that TEX Live builders do not compile it. Instead the author submits binaries. The `codesign` script didn't work with this binary. I contacted the author, Philip Kime. A month later he sent a binary which worked. I suspect Kime knows a lot more about notarization than I do now.

6.5 Case 4: The big enchilada

Finally it was time to notarize the full TEX Live. I hardened `xz`, `wget`, `lz4`, and all the binaries in `bin/x86_64-darwin` which were not links and reported to be "Mach-0 64-bit executables" by `file`. Tests revealed that two of these binaries needed an exception for X11: `mf` and `xdvi-xaw`. I submitted the package to Apple. It was rejected.

A big difference between BasicTEX and the full TEX Live is that the second package has documentation provided by package makers. This documentation comes in a wide variety of formats: source files for illustrations, zip files, and so forth. When Apple tests an install package for viruses, does it unzip files and look inside? Yes, it does. Does it examine illustration source files? Yes, it does that too. So lots of things could go wrong.

Luckily, Apple provided clear explanations for rejection, and it turned out that MacTEX had only three problems:

- In `texmf-dist/doc/support/ctan-o-mat`, one file is given an extension `.pkg`. Apple believes that a file with extension `.pkg` is an install package, and this package was not signed. It turned out to be an ordinary text file.
- In `texmf-dist/doc/latex/codepage`, Apple could not unzip the file `demo.zip`.
- In `texmf-dist/source/latex/stellenbosch`, there is a zip file named `USlogos-4.0-src.zip` containing two CorelDraw source files for illustrations. Apple did not recognize these source files and flagged them.

The three problems were easy to work around. Bug reports were also sent to Apple so they can improve the notarization machinery.

7 Status of notarization for MacTEX-2019

Fully notarized install packages for MacTEX-2019, BasicTEX-2019, and Ghostscript-9.27 are available on the web for testing. Indeed, the Ghostscript-9.27 package on CTAN is already notarized. The MacTEX-2019 and BasicTEX-2019 packages will be moved to CTAN, replacing the original packages, in late summer just before Catalina is released.

TEX Live Utility, LATEXiT, and BibDesk are not in the notarized MacTEX-2019 because they are applications rather than command line programs, *so their authors must sign and notarize them.* This has not yet happened. If these authors used the XCode which comes with Mojave, these steps would be trivial, but they use an older XCode. We are working with the authors but have nothing to report.

8 Technical details

I end with some technical details for others who may need to deal with these issues on the Macintosh. I'll explain how to sign install packages and how to notarize such packages. Then I'll list the six runtime entitlements and seven resource access entitlements from an official Apple document.

8.1 Signing an install package

Signing requires developer status from Apple, which costs $100 a year. Certificate information and security codes are kept on Apple's KeyChain, and automatically retrieved by the signing software when needed. If you buy a new machine or install a new system, you must transfer this information to the new system. XCode makes this easy *if* you know what mysterious icon to click.

Signing applications happens automatically in XCode as part of the build process. Signing install packages is done on the command line. The command here signs `Temp.pkg` and writes the signed package `Basic.pkg`.

```
productsign  \
 --sign "Developer ID Installer: Richard Koch" \
 Temp.pkg Basic.pkg
```

8.2 Notarizing an install package

Notarization of install packages is done on the command line, and is somewhat trickier. Below are the crucial commands. The first command sends an install package to Apple to be notarized. If uploading succeeds, this command returns an identifier which I symbolize with `YYYY`; it is actually much longer.

```
xcrun altool --notarize-app \
  --primary-bundle-id \
       "org.tug.mactex.basictex" \
--username "koch@uoregon.edu" \
--password "XXXX" \
--file BasicTeX.pkg
```

When Apple is finished, it sends a brief email stating whether notarization was successful. If there were errors, this second command asks for a detailed list of errors. The command returns a url, and the error list will then appear in a browser pointed to this url.

```
xcrun altool --notarization-info YYYY \
  --username  "koch@uoregon.edu" \
  --password "XXXX"
```

If notarization was successful, this third command staples the certificate to the install package, producing a notarized package:

```
xcrun stapler staple "BasicTeX.pkg"
```

In these commands, `altool` is a command line tool which communicates with Apple. This communication is normally protected using two-factor authentication, but that is not convenient for command line work. So before using `altool`, Apple asks developers to log into their account and give `altool` a temporary password. The symbol `XXXX` in the first and second commands represents this password.

The value `org.tug.mactex.basictex` in the first command identifies the install package for the notification process, but need not correspond to any similar string in the package. So the identifier can be selected randomly.

8.3 Runtime entitlements

All entitlements are boolean values; all keys start with `com.apple.security`, not shown here for brevity.

Allow Execution of JIT-compiled Code: whether the app may create writable and executable memory using the `MAP_JIT` flag. Key: `.cs.allow-jit`

Allow Unsigned Executable Memory: whether the app may create writable and executable memory without using the `MAP_JIT` flag. Key: `.cs.allow-unsigned-executable-memory`

Allow DYLD Environment Variables: whether the app may be impacted by DYLD environment variables, which can be used to inject code into the process. Key: `.cs.allow-dyld-environment-variables`

Disable Library Validation: whether the app may load plug-ins or frameworks signed by other developers. Key: `.cs.disable-library-validation`

Disable Executable Memory Protection: whether to disable code signing protections while launching the app. Key: `.cs.disable-executable-page-protection`

Debugging Tool: whether the app is a debugger and may attach to other processes or get task ports. Key: `.cs.debugger`

8.4 Resource access entitlements

Audio Input: whether the app may record audio using the built-in microphone and access audio input using Core Audio. Key: `.device.audio-input`

Camera: whether the app may capture movies and still images using the built-in camera. Key: `.device.camera`

Location: whether the app may access location information from Location Services. Key: `.personal-information.location`

Address Book: whether the app may have read-write access to contacts in the user's address book. Key: `.personal-information.addressbook`

Calendars: whether the app may have read-write access to the user's calendar. Key: `.personal-information.calendars`

Photos Library: whether the app may have read-write access to the user's Photos library. Key: `.personal-information.photos-library`

Apple Events: whether the app may send Apple Events to other apps. Key: `.automation.apple-events`

⋄ Richard Koch
koch (at) math dot uoregon dot edu
http://math.uoregon.edu/koch/

Quickref: Common Lisp reference documentation as a stress test for Texinfo

Didier Verna

Abstract

Quickref is a global documentation project for the Common Lisp ecosystem. It creates reference manuals automatically by introspecting libraries and generating corresponding documentation in Texinfo format. The Texinfo files may subsequently be converted into PDF or HTML. Quickref is non-intrusive: software developers do not have anything to do to get their libraries documented by the system.

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Common Lisp ecosystem at once. The result is a website containing almost two thousand reference manuals. Quickref provides a Docker image for an easy recreation of this website, but a public version is also available and actively maintained.

Quickref constitutes an enormous and successful stress test for Texinfo. In this paper, we give an overview of the design and architecture of the system, describe the challenges and difficulties in generating valid Texinfo code automatically, and put some emphasis on the currently remaining problems and deficiencies.

1 Introduction

Lisp is a high level, general purpose, multi-paradigm programming language created in 1958 by John McCarthy [2]. We owe to Lisp many of the programming concepts that are still considered fundamental today (functional programming, garbage collection, interactive development, *etc.*). Over the years, Lisp has evolved as a family of dialects (including Scheme, Racket, and Clojure, to name a few) rather than as a single language. Another Lisp descendant of notable importance is Common Lisp, a language targeting the industry, which was standardized in 1994 [5].

The Lisp family of languages is mostly known for two of its most prominent (and correlated) characteristics: a minimalist syntax and a very high level of expressiveness and extensibility. The root of the latter, right from the early days, is the fact that code and data are represented in the same way (a property known as *homoiconicity* [1, 3]). This makes meta-programming not only possible but also trivial. Being a Lisp, Common Lisp not only maintains this property, but also provides an unprecedented arsenal of paradigms making it much more expressive and extensible than its industrial competitors such as C++ or Java.

Interestingly enough, the technical strengths of the language bring serious drawbacks to its community of programmers (a phenomenon affecting all the dialects). These problems are known and have been discussed many times [4, 7]. They may explain, at least partly, why in spite of its technical potential, the Lisp family of languages never really took over, and probably never will. The situation can be summarized as follows: Lisp usually makes it so easy to "hack" things away that every Lisper ends up developing his or her own solution, inevitably leading to a *paradox of choice*. The result is a plethora of solutions for every single problem that every single programmer faces. Most of the time, these solutions work, but they are either half-baked or targeted to the author's specific needs and thus not general enough. Furthermore, it is difficult to assert their quality, and they are usually not (well) documented.

As this situation is well known, the community has been attempting to "consolidate" itself in various ways. Several websites aggregate resources related to the language or its usage (books, tutorials, implementations, development environments, applications, *etc.*). The Common Lisp Foundation (`cl-foundation.org`) provides technical, sometimes even financial, support and infrastructure for project authors. Once a year, the European Lisp Symposium (`european-lisp-symposium.org`) gathers the international community, open equally to researchers and practitioners, newcomers and experts.

From a more technical standpoint, solving the paradox of choice, that is, deciding on official solutions for doing this or that, is much more problematic — there is no such thing as an official authority in the community. On the other hand, some libraries do impose themselves as *de facto* standards. Two of them are worth mentioning here. Most non-trivial Common Lisp packages today use ASDF for structuring themselves (fig.1 has an example). ASDF allows you to define your package architecture in terms of source files and directories, dependencies and other metadata. It automates the process of compiling and loading (dependencies included). The second one is Quicklisp (`quicklisp.org`). Quicklisp is both a central repository for Common Lisp libraries (not unlike CTAN) and a programmatic interface for it. With Quicklisp, downloading, installing, compiling and loading a specific package on your machine (again, dependencies included) essentially becomes a one-liner.

One remaining problem is that of documentation. Of course, it is impossible to force a library author to properly document his or her work. One

```
(asdf:defsystem :net.didierverna.declt
  :long-name "Documentation Extractor from Common Lisp to Texinfo"
  :description "A reference manual generator for Common Lisp libraries"
  :author "Didier Verna"
  :mailto "didier@didierverna.net"
  :homepage "http://www.lrde.epita.fr/~didier/software/lisp/"
  :source-control "https://github.com/didierverna/declt"
  :license "BSD"
  ...)
```

Figure 1: ASDF system definition excerpt

could consider writing the manuals they miss for the third-party libraries they use, but this never happens in practice. There is still something that we can do to mitigate the issue, however. Because Common Lisp is highly reflexive, it is relatively straightforward to retrieve the information necessary to automatically create and typeset *reference* manuals (as opposed to *user* manuals). Several such projects exist already (remember the paradox of choice). In this paper we present our own, probably the most complete Common Lisp documentation generator to date.

Enter Quickref. . .

2 Overview

Quickref is a global documentation project for the Common Lisp ecosystem. It generates reference manuals for libraries available in Quicklisp automatically. Quickref is non-intrusive, in the sense that software developers do not have anything to do to get their libraries documented by the system: mere availability in Quicklisp is the only requirement. In this section, we provide a general overview of the system's features, design, and implementation.

2.1 Features

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Quicklisp world at once, which means that almost two thousand reference manuals are generated. Creating a local documentation website can be done in two different ways: either by using the provided Docker image (the most convenient solution for an exhaustive website), or directly via the programmatic interface, from within a running Lisp environment (when only the documentation for the local, partial, installation is required). If you don't want to run Quickref yourself, a public website is also provided and actively maintained at `quickref.common-lisp.net`. It always contains the result of a full run of the system on the latest Quicklisp distribution.

2.2 Documentation items

Reference manuals generated by Quickref contain information collected from various sources. First of all, many libraries provide a README file of some sort, which can make for a nice introductory chapter. In addition to source files and dependencies, ASDF offers ways to specify project-related metadata in the so-called *system definition* form. Figure 1 illustrates this. Such information can be easily (programmatically) retrieved and used. Next, Lisp itself has some built-in support for documentation, in the form of so-called *docstrings*. As their name suggests, docstrings are (optional) documentation strings that may be attached to various language constructs such as functions, variables, methods and so on. Figure 2 has an example. When available, docstrings greatly contribute to the completeness of reference manuals, and again, may be retrieved programmatically through a simple standard function call.

```
(defmacro @defconstant (name &body body)
  "Execute BODY within a @defvr Constant.
NAME is escaped for Texinfo prior to rendering.
BODY should render on *standard-output*."
  `(@defvr "Constant" ,name ,@body))
```

Figure 2: Common Lisp docstring example

As for the rest, the solution is less straightforward. We want our reference manuals to advertise as many software components as possible (functions, variables, classes, packages, *etc.*). In general there are two main strategies for collecting this information.

Code walking. The first one, known as *code walking*, consists of statically analyzing the source code. A code walker is usually at least as complicated as the syntax of the target language, because it requires a parser for it. Because of Lisp's minimalist syntax, using a code walker is a very tempting solution. On the other hand, Lisp is extremely dynamic in nature, meaning that many of the final program's components may not be directly visible in the source

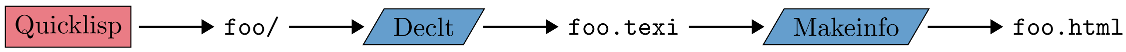


Figure 3: Quickref pipeline (Main thread, External Process)

code. On top of that, programs making syntactic extensions to the language would not be directly parsable. In short, it is practically impossible to collect all the required information by code walking alone. Therefore, we do not use that approach.

Introspection. Our preferred approach is by *introspection*. Here, the idea is to actually compile and load the libraries, and then collect the relevant information by inspecting memory. As mentioned before, the high level of reflexivity of Lisp makes introspection rather straightforward. This approach is not without its own drawbacks however. First, actually compiling and loading the libraries requires that all the necessary (possibly foreign) components and dependencies are available. This can turn out to be quite heavy, especially when the two thousand or so Quicklisp libraries are involved. Secondly, some libraries have platform, system, compiler, or configuration-specific components that may or may not be compiled and loaded, depending on the exact conditions. If such a component is skipped by our system, we won't see it and hence we won't document it. We think that the simplicity of the approach by introspection greatly compensates for the risk of missing a software component here and there. That is why introspection is our preferred approach.

2.3 Toolchain

Figure 3 depicts the typical manual production pipeline used by Quickref, for a library named `foo`.

1. Quicklisp is used first, to make sure the library is installed, which results in the presence of a local directory for that library.
2. Declt (`lrde.epita.fr/~didier/software/lisp/misc.php#declt`) is then run on that library to generate the documentation. Declt is another library of ours, written five years before Quickref, but with that kind of application in mind right from the start. In particular, it is for that reason that the documentation generated by Declt is in Texinfo intermediate format.
3. The Texinfo file is processed into HTML. Texinfo (`gnu.org/software/texinfo`) is the GNU official documentation format. There are three main reasons why this format was chosen when Declt was originally written. First, it is particularly well suited to technical documentation. More importantly, it is designed as an abstract, intermediate format from which human-readable documentation can in turn be generated in many different forms (notably PDF and HTML). Finally, it includes very convenient built-in anchoring, cross-referencing, and indexing capabilities.

Quickref essentially runs this pipeline on the required libraries. Some important remarks need to be made about this process.

1. Because Declt works by introspection, it would be unreasonable to load almost two thousand libraries in a single Lisp image. For that reason, Quickref doesn't actually run Declt directly, but instead forks it as an external process.
2. Similarly, `makeinfo` (`texi2any` in fact), the program used to convert the Texinfo files to HTML, is an external program written in Perl (with some parts in C), not a Lisp library. Thus, here again, we fork a `makeinfo` process out of the Quickref Lisp instance in order to run it.

2.4 Performance

Experimental studies have been conducted on the performance of the system. There are different scenarios in which Quickref may run, depending on the exact number of libraries involved, their current state, and the level of required "isolation" between them. All the details are provided in [6], but in short, there is a compromise to be made between the execution time and the reliability of the result. We found that for a complete sequential run of the system on the totality of Quicklisp, the most frequent scenario takes around two hours on our test machine, whereas the safest one requires around seven hours.

In order to improve the situation, we recently added support for parallelism to the system. The upgraded architecture is depicted in Figure 4. In this new processing scheme, an adjustable number of threads is devoted to generating the Texinfo files in parallel. In a second stage, a likewise adjustable number of threads is in charge of taking the Texinfo files as they come, and creating the corresponding HTML versions. A specific scheduling algorithm (not unlike that of the `make` program) delivers libraries in an order, and at a time suitable to parallel processing by the Declt threads, avoiding any concurrency problems. With this new architecture in place, we were able to cut the processing time by a factor of four, reducing the worst case scenario to 1h45 and the most frequent one to half an hour. These numbers

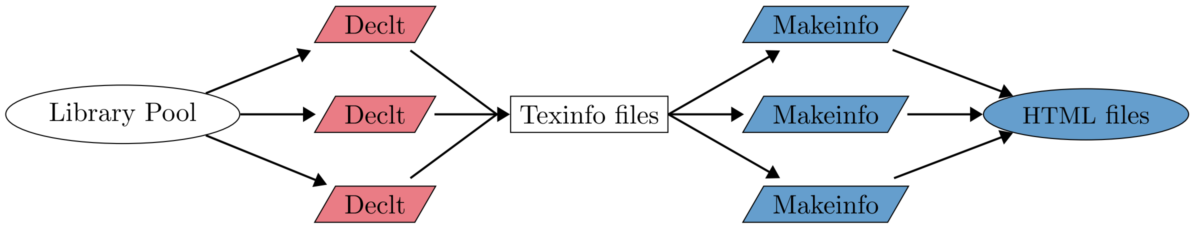


Figure 4: Quickref parallel processing (Declt thread, Makeinfo thread)

make it reasonable to run Quickref on one's local machine again.

3 Challenges

Quickref is a challenging project in many regards. Two thousand libraries is *a lot* to process. Setting up the environment necessary to properly compile and run those libraries is not trivial, especially because many of them have platform or system-specific code and require foreign dependencies. Finally, Quickref constitutes a considerable (and successful) stress test for Texinfo. The Texinfo file sizes range from 7KB to 15MB (double that for the generated HTML ones). The number of lines of Texinfo code in those files extends from 364 to 285 020, the indexes may contain between 14 and 44 500 entries, and the processing times vary from 0.3s to 1m 38s per file.

Challenges related to the project scalability and performance have been described previously [6]. This section focuses on more general or typesetting/Texinfo issues.

3.1 Metadata format underspecification

One difficulty in collecting metadata is that their format is often underspecified, or not specified at all, as is the case with ASDF system items. To give just one example, Figure 5 lists several of the possible values we found for the author metadata. As you can see, most programmers use strings, but the actual contents vary greatly (single or multiple names, email addresses, middle letter, nicknames, *etc.*), and so does the formatting. For the anecdote, we found one attempt at pretty printing the contents of the string with a history of authors, and one developer even went as far as concealing his email address by inserting Lisp code into the string itself...

It would be unreasonable to even try to understand all these formats (what others will we discover in the future?), so we remain somewhat strict in what we recognize — in this particular case, strings either in the form `"author"` or `"author <email>"` (as in either:

```
"Didier Verna"
"Didier Verna <didier@lrde.epita.fr>"
```

respectively), or a list of these. The Declt user manual has a Guidelines section with some advice for library authors that would like to be friendlier with our tool. We cannot force anyone to honor our guidelines however.

```
"Didier Verna"
"Didier Verna <didier@lrde.epita.fr>"
"Didier Verna didier@lrde.epita.fr"
"didier@lrde.epita.fr"
"<didier@lrde.epita.fr>"
"Didier Verna and Antoine Martin"
"Didier Verna, Antoine Martin"
"Didier Verna Antoine Martin"
"D. Verna Antoine E Martin"
"D. Verna Antoine \"Joe Cool\" Martin"
("Didier Verna" "Antoine Martin")
"
     Original Authors:
       Salvi Péter,
       Naganuma Shigeta,
       Tada Masashi,
       Abe Yusuke,
       Jianshi Huang,
       Fujii Ryo,
       Abe Seika,
       Kuroda Hisao
     Author Post MSI CLML Contribution:
       Mike Maul  <maul.mike@gmail.com>"
"(let ((n \"Christoph-Simon Senjak\"))
          (format nil \"~A <~C~C~C~C~A>\"
            n (elt n 0) (elt n 10) (elt n 16)
            #\\@ \"uxul.de\"))"
```

Figure 5: ASDF author metadata variations

On the other hand, Quickref has an interesting social effect that we particularly noticed the first time the public website was released. In general, people don't like *our* documentation for *their* work to look bad, especially when it is publicly available. In the first few days following the initial release and announcement of Quickref, we literally got dozens of reports related to typesetting glitches. Programmers *rushed* to the website in order to see what *their* library looked like. If the bugs were not on our side, many of the concerned authors were hence willing to slightly bend their own coding style, in order for *our* documentation to look better. We still count on that social effect.

3.2 Definitions grouping

Rather than just providing a somewhat boring list of functions, variables, and other definitions, as reference manuals do, Declt attempts to improve the presentation in different ways. In particular, it tries to group related definitions together when possible.

A typical example of this is when we need to document accessors (readers and writers to the same information). It makes sense to group these definitions together, provided that their respective docstrings are either nonexistent, or exactly the same (this is one of the incentives given to library authors in the Declt guidelines). This is exemplified in Figure 6. Another typical example consists in listing methods (in the object-oriented sense) within the corresponding generic function's entry.

`context-hyperlinksp` *CONTEXT* [Function]
`(setf context-hyperlinksp)` *BOOL CONTEXT* [Function]
Access *CONTEXT*'s hyperlinksp flag.

Package [`net.didierverna.declt`], page 29,
Source [`doc.lisp`], page 24, (file)

Figure 6: Accessors definitions grouping

Texinfo provides convenient macros for defining usual programming language constructs (`@defun`, `@defvar`, *etc.*), and "extended" versions for adding sub-definitions (`@defunx`, `@defvarx`, *etc.*). Unfortunately, definitions grouping prevents us from using them, for several reasons.

1. Nesting `@def...` calls would lead to undesirable indentation.
2. Heterogeneous nesting is prohibited. For example, it is not possible use `@defvarx` within a call to `@defun` (surprising as it may sound, this kind of heterogeneous grouping makes sense in Lisp).

On the other hand, that kind of thing is possible with the lower-level (more generic) macros, as heterogeneous *categories* become simple macro arguments. One can, for example use the following (which we frequently do):

```
@deffn {Function} ...
@deffnx {Compiler Macro} ...
...
@end deffn
```

This is why we stick to those lower-level macros, at the expense of re-inventing some of the higher-level built-in functionality.

Even with this workaround, some remaining limitations still get in our way.

1. There are only nine canonical categories and it is not possible to add new ones (at least not without hacking Texinfo's internals).
2. Although we understand the technical reasons for it (parsing problems, probably), some of the canonical categories are arguable. For example, the distinction between typed and untyped functions makes little sense in Common Lisp which has optional static typing. We would prefer to have a single function definition entry point handling optional types.
3. Heterogeneous mixing of the lower-level macros is still prohibited. For example, it remains impossible to write the following (still making sense in Lisp):

   ```
   @deffn {Function} ...
   @defvrx {Symbol Macro} ...
   ...
   @end deffn
   ```

3.3 Pretty printing

Pretty printing is probably the biggest challenge in typesetting Lisp code, because of the language's flexibility. In particular, it is very difficult to find the right balance between readability and precision.

Identifiers. In Lisp, identifiers can be basically *anything*. When identifiers contain characters that are normally not usable (*e.g.* blanks or parentheses), the identifier must be escaped with pipes. In order to improve the display of such identifiers, we use several heuristics.

- A symbol containing blank characters is normally escaped like this: `|my identifier|`. Because the escaping syntax doesn't look very nice in documentation, we replace blank characters with more explicit Unicode ones, for instance `my␣identifier`. We call this technique "revealing". Of course, if one identifier happens to contain one of our revealing characters already, the typesetting will be ambiguous. This case is essentially nonexistent in practice, however.
- On the other hand, in some situations it is better to *not* reveal the blank characters. The so-called *setf* (setter / writer) functions are such an example. Here, the identifier is in fact composed of several symbols, such as in `(setf this)`. Revealing the whitespace character would only clutter the output, so we leave it alone.
- Finally, some unusual identifiers that are normally escaped in Lisp, such as `|argument(s)|`,

do not pose any readability problems in documentation, so we just typeset them without the escaping syntax.

Qualification. Another issue is symbol *qualification*. With one exception, symbols in Lisp belong to a *package* (more or less the equivalent of a namespace). Many Lispers use Java-style package names, which can end up being quite long. Typesetting a fully qualified symbol would give something like this: `my.long.package.name:symbol`. Lisp libraries usually come with their own very few packages, so typesetting a reference manual with thousands of symbols fully qualified with the same package name would look pretty bad. Because of that, we avoid typesetting the package names in general. Unfortunately, if different packages contain eponymous symbols, this leads to confusing output. Currently, we don't have a satisfactory answer to this problem.

Docstrings. The question of how to typeset docstrings is also not trivial. People tend to use varying degrees of plain-text formatting in them, with all kinds of line lengths, *etc.* Currently, we use only a very basic heuristic to determine whether an end of line in a docstring is really wanted here, or just a consequence of reaching the "right margin". We are also considering providing an option to simply display the docstrings verbatim. In the long term, we plan to support markup languages such as Markdown.

References. A Texinfo-related problem we have is that links are displayed differently, depending on the output format, and with some rather undesirable DWIM behavior. Table 1 shows the output of a call to `@ref{anchor, , label}` in various formats (`anchor` is the link's internal name, `label` is the desired output).

Table 1: Texinfo links formatting in various output formats

HTML	label
PDF	[label], page 12,
Info	*note label: anchor.
Emacs Info mode	See label.

In PDF, the presence of the trailing comma is context dependent. In Info, both the label and the actual anchor name are typeset, which is very problematic for us (see Section 3.4). In Emacs Info mode, the casing of "See" seems to vary. In general, we would prefer to have more consistent output across the different formats, or at least, more control over it.

3.4 Anchoring

The final Texinfo challenge we want to address here is that of anchoring. In Texinfo, anchor names have severe limitations: dots, commas, colons, and parentheses are explicitly forbidden (due to the final display syntax in Info). This is very unfortunate because those characters are extremely common in Lisp (parentheses of course, but also dots and colons in the package qualification syntax).

Note that formats other than Info are not affected by this problem. There is an Info-specific workaround documented in Appendix G.1 of the Texinfo user manual. In short, a sufficiently recent version can automatically "protect" problematic node names by surrounding them with a special marker in the resulting Info files. Unfortunately, neither older Info readers, nor the current Emacs mode are aware of this feature. Besides, the latest stable release of Texinfo still has problems with it (menus do not work correctly). Consequently, this workaround is not a viable solution for us (yet).

Our original (and still current) solution is to replace those characters by a sequence such as `<dot>`. Of course, this makes anchor names particularly ugly, but we didn't think that was a problem because we have nicer *labels* to point to them in the output (in fact, labels have a less limited syntax, although this is not well documented). However, we later realized that anchor names still appear in the HTML output and also in pure Info. Consequently, we are now considering changing our escaping policy, perhaps by using Unicode characters as replacements, just as we already do on identifiers (see Section 3.3).

The second anchoring problem we have is that of Texinfo nodes, the fundamental document structuring construct. In addition to the aforementioned restrictions related to anchoring, nodes have two very strong limitations: their names must be unique and there is no control over the way they are displayed in the output. This is a serious problem for us because Lisp has a lot of different namespaces. A symbol may refer to a variable, a function, a class, and many other things at the same time. Consequently, when nodes are associated with Lisp symbols, we need to mangle their names in a way that makes them barely human readable. Because of that, our use of nodes remains rather limited, which is somewhat paradoxical, given the importance of nodes in Texinfo. Apart from general, high level sectioning, the only nodes associated with Lisp symbols are for ASDF components and packages, probably already a bit too much. It is our hope that one day, the node names uniqueness constraint in Texinfo might be relaxed, perhaps disambiguating by using their hierarchical organization.

4 Conclusion and perspectives

Although a relatively young project, Quickref is already quite successful. It is able to document almost two thousand Common Lisp libraries without any showstoppers. Less than 2% of the Quicklisp libraries still pose problems and some of the related bugs have already been identified. The Common Lisp community seems generally grateful for this project.

Quickref also constitutes an enormous, and successful, stress test for Texinfo. Given the figures involved, it was not obvious how `makeinfo` would handle the workload, but it turned out to be very reliable and scalable. Although the design of Texinfo sometimes gets in our way, we still consider it a good choice for this project, in particular given the diversity of its output formats and its built-in indexing capabilities.

In addition to solving the problems described in this paper, the project also has much room for improvement left. In particular, the following are at the top level of our TODO list.

1. The casing problem needs to be addressed. Traditional Lisp is case-insensitive but internally upcases every symbol name (except for escaped ones). Several modern Lisps offer alternative policies with respect to casing. Quickref doesn't currently address casing problems at all (not even that of escaped symbols).
2. Our indexing policy could be improved. Currently, we only use the built-in Texinfo indexes (Functions, Variables, *etc.*) but we also provide one level of sub-indexing. For instance, macros appear in the function index, but they are listed twice: once as top level entries, and once under a Macro sub-category. The question of which amount of sub-indexing we want, and whether to create and use new kinds of indexes is under consideration.
3. Although our reference manuals are already stuffed with cross-references, we plan to add more. Because Declt was originally designed to generate one reference manual at a time, only internal cross-references are available. The existence of Quickref now raises the need for external cross-references (that is, between different manuals).
4. Many aspects of the pretty printing could be improved, notably that of so-called "unreadable" objects and lambda lists.
5. In addition to HTML, we plan to provide PDF as well as Info files on the website, since they are readily available.
6. We intend to integrate Quickref with Emacs and Slime (a *de facto* standard Emacs-based development environment for Common Lisp). In particular, we want to give Emacs the ability to browse the Info reference manuals online or locally if possible, and provide Slime with commands for opening the Quickref documentation directly from Lisp source code displayed in Emacs buffers.
7. Finally, we are working on providing new index pages for the website. Currently, we have a library index and an author index. We are working on providing keyword and category indexes as well.

References

[1] A. C. Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.

[2] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3(4):184–195, Apr. 1960. `doi:10.1145/367177.367199`

[3] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM* 3:214–220, Apr. 1960. `doi:10.1145/367177.367223`

[4] M. Tarver. The bipolar Lisp programmer. `marktarver.com/bipolar.html`, 2007.

[5] ANSI. American National Standard: Programming Language — Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[6] D. Verna. Parallelizing Quickref. In *12th European Lisp Symposium*, pp. 89–96, Genova, Italy, Apr. 2019. `doi:10.5281/zenodo.2632534`

[7] R. Winestock. The Lisp curse, Apr. 2011. `winestockwebdesign.com/Essays/Lisp_Curse.html`

◇ Didier Verna
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre
France
`didier (at) lrde dot epita dot fr`
`http://www.didierverna.info`

Combining LaTeX with Python

Uwe Ziegenhagen

Abstract

Even older than Java, Python has achieved a lot of popularity in recent years. It is an easy-to-learn general purpose programming language, with strong capabilities, including in state-of-the-art topics such as machine learning and artificial intelligence. In this article we want to present scenarios where LaTeX and Python can work jointly. We will show examples where LaTeX documents are automatically generated by Python or receive content from Python scripts.

1 Introducing Python

Python has steadily grown to be one of the most widely used programming languages. Invented in 1991 by Guido van Rossum at the Centrum Wiskunde & Informatica in the Netherlands, Version 1.0 appeared in 1994. The current versions are 2.7 and 3.x. For people who wish to start with Python, Python 3 is strongly recommended.

```
print('Hello' + ' ' + 'World')
```

Listing 1: The unavoidable "Hello World" example

Python has a strong emphasis on code readability by making whitespace significant. In contrast to other programming languages, Python uses whitespace and indentation to define code blocks; a first example is in Listing 2.

```
def addTwo(a, b):
    return a+b

print(addTwo(5,3))     # gives 8
print(addTwo('U','S')) # gives 'US'
```

Listing 2: Basic function definition example

Python supports various programming paradigms, such as procedural, object-oriented and functional programming. Listing 3 shows an example for the functional programming paradigm, using a lambda function to filter those integers from a list that are divisible by 2.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
result = filter(lambda x: x % 2 == 0, my_list)
print(list(result))
```

Listing 3: Using functional programming to filter a list

Listing 4 shows an example for the OO-programming paradigm. Here we define a class with two properties that is then instantiated.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_age(self):
        print(self.name + ', ' + str(self.age))

john = Person('John', 50)
john.print_age()
```

Listing 4: Using object-oriented programming

Excellent literature is available for Python learners on- and offline; we can recommend [1].

2 Writing LaTeX files with Python

After that brief introduction we will now focus on the creation of LaTeX files using Python. The recommended approach is to use a so-called "context managers", as it will handle the management of the file references as well as errors in case the file is not accessible or writable.

Listing 5 shows an example on how to write a simple LaTeX file. Backslashes need to be escaped, the line endings need to be added. Depending on the platform the code is executed, they will be replaced by the system's line ending. The resulting file is then UTF-8-encoded and can easily be processed further.

```
with open('sometexfile.tex','w') as file:
    file.write('\\documentclass{article}\n')
    file.write('\\begin{document}\n')
    file.write('Hello Palo Alto!\n')
    file.write('\\end{document}\n')
```

Listing 5: Writing a TeX file

Processing, e. g., the compilation by pdfLaTeX and display by the system's PDF viewer can also be triggered from Python, as Listing 5 shows. We create the LaTeX file and use Python's subprocess module to call pdfLaTeX. When this process has a non-error exit code, the platform's PDF viewer is launched.

```
import subprocess, os
with open('sometexfile.tex','w') as file:
    file.write('\\documentclass{article}\n')
    file.write('\\begin{document}\n')
    file.write('Hello Palo Alto!\n')
    file.write('\\end{document}\n')

x = subprocess.call('pdflatex sometexfile.tex')
if x != 0:
    print('Exit-code not 0, check result!')
else:
    os.system('start sometexfile.pdf')
```

Listing 6: Writing & processing TeX files

When LaTeX files are created programmatically the goal is often to create bulk letters or other dynamically adjusted documents. Python offers various ways to assist in this process. The most intuitive way is probably to use search & replace to eplace placeholders with text; Listing 7 shows an example for this approach. The example should be self-explaining, note the nested context managers to read and then write the LaTeX file.

```
place = 'Palo Alto'

with open('place.tex','r') as myfile:
    text = myfile.read()
    text_new = text.replace('$MyPlace$', place)

    with open('place_new.tex', 'w') as output:
        output.write(text_new)
```

Listing 7: Replacing text

While this approach works fine, it is not recommended when more complicated documents need to be created. Fortunately Python offers a variety of template engines — either built-in or easily installable with the help of Python's package manager — that improve the workflow and avoid "re-inventing the wheel". Among the different template engines, we have successfully worked with Jinja2. It offers full Unicode support, sandboxed execution, template inheritance and many more useful features. Listing 8 shows a non-LaTeX example for Jinja2, which tells us the following:

1. Syntax is (easily) understandable
2. Jinja2 brings its own notation for looping, etc.
3. Extensive use of `{`, `%`, `}`

```
from jinja2 import Template

mytemplate = Template("Hello {{place}}!")
print(mytemplate.render(place="Palo Alto"))

mytemplate = Template("Some numbers: {% for n
    in range(1,10) %}{{n}}{% endfor %}")
print(mytemplate.render())
```

Listing 8: A non-LaTeX Jinja2 template example

So, to make Jinja2 work well with LaTeX we need to modify the way a template is defined. Listing 2 shows[1] how this reconfiguration can be made. Instead of braces, we use two LaTeX commands, `\BLOCK` and `\VAR`. Both commands will later be defined as empty LaTeX commands in the LaTeX file to have the file compile without errors.

[1] Source: https://web.archive.org/web/20121024021221/http://e6h.de/post/11/

```
import os
import jinja2 as j

latex_env = j.Environment(
   block_start_string = '\BLOCK{',
   block_end_string = '}',
   variable_start_string = '\VAR{',
   variable_end_string = '}',
   comment_start_string = '\#{',
   comment_end_string = '}',
   line_statement_prefix = '%-',
   line_comment_prefix = '%#',
   trim_blocks = True,
   autoescape = False,
   loader = j.FileSystemLoader(os.path.abspath('.'))
   )
```

The following Listing 9 shows an excerpt from the final code. It loads the template, fills the placeholders and writes the final document to the disk. One advantage of this approach is that it allows the template to be separated from the program logic that fills it; in more complex situations, the built-in scripting comes very handy.

```
template = latex_env.get_template('jinja-01.tex')
document = template.render(place='Palo Alto')
with open('final-02.tex','w') as output:
    output.write(document)
```

Listing 9: Rendering the document

3 Running Python from LaTeX

In this section we want to address the reverse: not the creation of LaTeX code but the execution of Python code from within LaTeX. Several packages and tools are available to support this. Here we want to demonstrate two of them. One is derived from code posted to tex.stackexchange.com, the other, `pythontex`, is a well-maintained LaTeX package.

The idea for the code given below came from the fact, that LaTeX is a) able to write the content of environments to external files and b) is able to run external commands when `--shell-escape` is enabled. One just needs need to combine both to write and run external files. Based on our question on TSX, an easily implementable solution was given;[2] it is shown in Listing 10. When Python code is placed in a `pycode` environment inside a document, LaTeX writes the code to the filename specified in the parameter of the environment, runs Python on this file and pipes its output to a `.plog` file. This `.plog` file is then read by LaTeX and typeset with syntax highlighting provided by the `minted` package (which also uses Python internally).

The advantage of this approach is that it can be adjusted easily to different external programs, as

[2] https://tex.stackexchange.com/questions/116583

long as they are able to run in batch mode. One can easily adjust the way the code is included, e.g., we have worked successfully with a two-column setup in Beamer, where the left column shows the source code and the right column the result of the code execution. One disadvantage is that the programs are executed each time the LaTeX code is compiled.

```
\usepackage{minted}
\setminted[python]{frame=lines, framesep=2mm,
     baselinestretch=1.2, bgcolor=colBack,
     fontsize=\footnotesize, linenos}
\setminted[text]{frame=lines, framesep=2mm,
     baselinestretch=1.2, bgcolor=colBack,
     fontsize=\footnotesize, linenos}

\usepackage{fancyvrb}
\makeatletter
\newenvironment{pycode}[1]%
  {\xdef\d@tn@me{#1}%
   \xdef\r@ncmd{python #1.py > #1.plog}%
   \typeout{Writing file #1}%
   \VerbatimOut{#1.py}%
  }%
  {\endVerbatimOut %
   \toks0{\immediate\write18}%
   \expandafter\toks\expandafter1%
     \expandafter{\r@ncmd}%
     \edef\d@r@ncmd{\the\toks0{\the\toks1}}%
   \d@r@ncmd
   \noindent Input
   \inputminted{python}{\d@tn@me.py}%
   \noindent Output
   \inputminted{text}{\d@tn@me.plog}%
  }%
\makeatother
```

Listing 10: The `pycode` environment

The `pythontex` package [2] uses a more advanced approach: it can detect if the Python code has been edited or not. Only if an edit took place is the Python code rerun, thus saving time especially with more complicated Python code. The workflow is the following: first the LaTeX engine of your choice is run, followed by the `pythontex` executable, followed by another `latex` run. The package offers various LaTeX commands and corresponding environments; see the package documentation.

Let us show with an example (Listing 11) how the package can be applied. After loading the package `pythontex` we use the `\pyc` command, which only executes code and does not typeset it, for the first line of Python code. Here we instruct Python to load a function from the `yahoo_fin` library which allows us to retrieve stock information from Yahoo, given that an Internet connection is available.

In the following table we then use `\py` commands to specify which stock quote to be retrieved. This command requires the executed Python code to return a single expression.

```
\documentclass[12pt]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{pythontex}
\usepackage{booktabs}
\begin{document}

\pyc{from yahoo_fin import stock_info as si}

\begin{tabular}{lr}
\toprule
Company & Latest quote \\
\midrule
Apple  & \py{round(si.get_live_price("aapl"),2)} \\
Amazon & \py{round(si.get_live_price("amzn"),2)} \\
Facebook & \py{round(si.get_live_price("fb"),2)} \\
\bottomrule
\end{tabular}
\end{document}
```

Listing 11: Using `pythontex` to retrieve stock prices

Company	Latest quote
Apple	203.43
Amazon	1832.89
Facebook	190.16

Figure 1: Output resulting from Listing 11

The `pythontex` package provides many more features, among them even symbolic computation. It can thus be highly recommended.

4 Summary

We have shown how easy LaTeX documents can be enriched by Python, a scripting language that is easy to learn and fun to work with. Accompanying this article is the more extensive presentation held at TUG 2019, for which the interested reader is directed to the slides at `www.uweziegenhagen.de`.

References

[1] M. Lutz. *Learning Python.* O'Reilly, 2013.

[2] G. M. Poore. PythonTeX: Reproducible documents with LaTeX, Python, and more. *Comput. Sci. Disc.* 8(1), 2015. `ctan.org/pkg/pythontex`

⋄ Uwe Ziegenhagen
Escher Str. 221
50739 Cologne, Germany
ziegenhagen (at) gmail dot com
www.uweziegenhagen.de

Parsing complex data formats in LuaTeX with LPEG

Henri Menke

Abstract

Although it is possible to read external files in TeX, extracting information from them is rather difficult. Ad hoc solutions tend to use nested if statements or regular expressions provided by several macro packages. However, these quick hacks don't scale well and quickly become unmaintainable.

LuaTeX comes to the rescue with its embedded LPEG library for Lua. LPEG provides a domain-specific embedded language that allows for writing grammars in a natural way. In this article I give a quick introduction to Parsing Expression Grammars (PEG) and then show how to write simple parsers in Lua with LPEG. Finally we will build a JSON parser to demonstrate how easy it is to even parse complex data formats.

1 Quick introduction to LPEG and Lua

The LPEG library [1] is an implementation of Parsing Expression Grammars (PEG) for the Lua language. It provides a domain-specific embedded language for this task. Its domain is, naturally, parsing. It is embedded in Lua using overloading of arithmetic operators to give it a natural syntax. The language it implements is PEG. The LPEG library has been included in LuaTeX since the beginning [2]. The examples in this article are based on the talk "Using Spirit X3 to Write Parsers" which was given by Michael Caisse at CppCon 2015 [3], where the speaker introduces the Spirit X3 library for C++ to write parsers using PEG. The Spirit library is not too dissimilar from LPEG and if you are looking for a parser generator for C++, I recommend it.

To make sure that we are all on the same page and the reader can easily understand the syntactic constructions used throughout this manuscript, we review some aspects of the Lua language. First of all, let's note that all variables are global by default, whereas local variables have to be preceded by the `local` keyword.

```
local x = 1
```

Most of the time we want definitions to be scoped, so this pattern will show up often. Another important thing to note about the Lua language is that, in contrast to many other programming languages, functions are first class variables. That means that when we declare a function, what we actually do is assign a value of type `function` to a variable. That is to say, these two statements are equivalent:

```
function f(...) end
f = function(...) end
```

Lua implements only a single complex data structure, the table. Tables in Lua act as both arrays and key–value storage; in fact, it is possible to mix both forms of access within a single instance, as in the following:

```
local t = { 11, 22, 33, foo = "bar" }
print(t[2], t["foo"], t.foo) -- 22 bar bar
```

As can be inferred from that, array indexing in Lua starts at `1`. For tables and strings Lua offers a useful shortcut. When calling a function with a single literal string or table, parentheses can be omitted. In the following snippet the statements on the left are equivalent to the ones on the right.

```
f("foo")            f"foo"
f({ 11, 22, 33 })   f{ 11, 22, 33 }
```

Especially when programming with LPEG this shortcut can save a lot of typing and, once used to it, makes the code a lot more readable. I will make extensive use of this syntax.

2 Why use PEG?

Before we delve into the inner workings of LPEG, let me first give some motivation as to why we would like to build parsers using PEG. Imagine trying to verify that input has a certain format, e.g. a date in the form day-month-year: `09-08-2019`. One approach could be to split the input at the hyphens and verify that each field only contains numbers, which is simple enough to implement using TeX macro code. However, the task quickly becomes more complicated when further requirements come into play. Merely because something is made up of three groups of numbers doesn't make it a valid date. In situations like these, regular expressions (regex) sound like a good solution and in fact, the regex to parse a "valid" date looks fairly innocent:

```
[0-3][0-9]-[0-1][0-9]-[0-9]{4}
```

I put "valid" in quotation marks, because obviously this regex misses several cases, such as different number of days in different months or leap years. I encourage the reader to look up a regular expression which covers these special cases, to get an impression as to how quickly the regex gets out of hand. Furthermore, neither a pure TeX solution nor regex implementations in TeX are fully expandable, which is often desirable. Maybe they can be made fully expandable but not without tremendous effort.

3 What is PEG?

The question remains, how does PEG help us here? Let's first look at a more or less formal definition of PEG, adapted from Wikipedia [4]. A parsing expression grammar consists of:

- A finite set N of non-terminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of parsing rules.
- An expression e_S termed the starting expression.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a parsing expression.

To illustrate this, we have a look at the following imaginary PEG for an email address.

$$\langle\text{name}\rangle \leftarrow \texttt{[a-z]+}\ (\texttt{"."}\ \texttt{[a-z]+})^*$$
$$\langle\text{host}\rangle \leftarrow \texttt{[a-z]+}\ \texttt{"."}\ (\texttt{"com"/"org"/"net"})$$
$$\langle\text{email}\rangle \leftarrow \langle\text{name}\rangle\ \texttt{"@"}\ \langle\text{host}\rangle$$

The symbols in angle brackets are the non-terminal symbols. The quoted strings and expressions in square brackets are terminal symbols. The entry point e_S is the rule named email (not specially marked).

The present grammar translates into natural language rather nicely. We start at the entry point, the email rule. The email rule tells us that an email is a name, followed by a literal `@`, followed by a host. The symbols name and host are non-terminal, meaning they can't be parsed without further information, so we have to resolve them. A name is specified as one or more characters in the range `a` to `z`, followed by zero or more groups of a literal dot, followed by one or more characters `a` to `z`. A host is one or more characters `a` to `z`, followed by a literal dot, followed by one of the literals `com`, `org`, or `net`. Here the range of characters and the string literals are terminal symbols, because they can be parsed from the input without further information.

As a little teaser, we will have a look at how the above grammar translated into LPEG.

```
local name  = R"az"^1 * (P"." * R"az"^1)^0
local host  = R"az"^1 * P"."
            * (P"com" + P"org" + P"net")
local email = name * P"@" * host
```

We can already see that there is some sort of mapping to translate PEG into LPEG; indeed, at first sight it seems like this translation is almost 1:1. We will learn what the symbols mean in the next section.

4 Basic parsers

LPEG provides some basic parsers to make life a little easier. These map the terminal symbols in the grammar. Here they are, with examples:

- `lpeg.P(string)` Matches `string` exactly. This matches “hello” but not “world”:

  ```
  lpeg.P("hello")
  ```

- `lpeg.P(n)` Matches exactly `n` characters. To match any single character we could use

  ```
  lpeg.P(1)
  ```

  There is a special character which is not mapped by any encoding — the end of input. In LPEG there is a special rule for this:

  ```
  lpeg.P(-1)
  ```

- `lpeg.S(string)` Matches any character in `string` (a set). To match normal whitespace we could use:

  ```
  lpeg.S(" \t\r\n")
  ```

- `lpeg.R("xy")` Matches any character between `x` and `y` (a range). To match any digit:

  ```
  lpeg.R("09")
  ```

  To match any character in the ASCII range we can combine lowercase and uppercase letters:

  ```
  lpeg.R("az", "AZ")
  ```

It is tedious to constantly type the `lpeg.` prefix, so we omit it from now on. This can be achieved by assigning the members of the `lpeg` table to the corresponding variables.

```
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R -- etc.
```

5 Parsing expressions

By themselves these basic parsers are rather useless. The real power of LPEG comes from the ability to arbitrarily combine parsers. This is achieved by means of parsing expressions. The available parsing expressions are listed in table 1. Below, I show some examples where the quoted strings in the comments represent input that is parsed successfully by the associated parser unless stated otherwise.

- Sequence: This implements the “followed by” operation, i.e. the parser matches only if the first pattern is followed directly by the second pattern.

Description	**PEG**	**LPEG**
Sequence	$e_1 e_2$	`patt1 * patt2`
Ordered choice	$e_1 \| e_2$	`patt1 + patt2`
Zero or more	e^*	`patt^0`
One or more	$e+$	`patt^1`
Optional	$e?$	`patt^-1`
And predicate	$\&e$	`#patt`
Not predicate	$!e$	`-patt`
Difference		`patt1 - patt2`

Table 1: Available parsing expressions in LPEG with their name and corresponding symbol in PEG. Note that the difference operator is an extension in LPEG and not available in PEG.

```
P"pizza" * R"09" -- "pizza4"
P(1) * P":" * R"09" -- "a:9"
```

- Ordered choice: The ordered choice parses the first operand first and only if it fails continues to the next operand. So the ordering is indeed important.

```
R"az" + R"09" + S".,;:?!"
-- "a", "9", ";"
-- "+" fails to parse
```

- Zero or more, one or more, and optional: These are all captured by the same construct in LPEG, the exponentiation operator. A positive exponent n parses at least n occurrences of the pattern, a negative exponent $-n$ parses at most n occurrences of the pattern.

```
R"az"^0 + R"09"^1
-- "z86", "abcde99", "99"
R"az"^1 + R"09"^1
-- "z86", "abcde99"
-- "99" fails to parse
R"az"^-1 + R"09"^1
-- "z86", "99"
-- "abcde99" fails to parse
```

- And predicate, not predicate: These are special in that they do not consume any input. As might be expected, the not predicate only matches if the parser it negates does not match.

```
R"09"^1 * #P";"
-- "86;"
-- "99" fails to parse
P"for" * -(R"az"^1)
-- "for()"
-- "forty" fails to parse
```

- Difference: The difference operator matches the first operand only if the second operand does not match. This can be useful to match C style comments which collect everything between the first `/*` and the first `*/`. However, care must be taken that the second operand cannot successfully parse parts of the first operand. If that is the case, the resulting rule will never match.

```
P"/*" * (1 - P"*/")^0 * P"*/"
-- "/* comment */"
P"helloworld" - P"hell"
-- will never match!
```

6 Simple examples

Let us study a simple example which parses two words separated by a space. The LPEG grammar is stored in the variable `rule`. The rest of the example shows the boilerplate that is necessary.

```
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R

local input = "cosmic pizza"

local rule = R"az"^1 * P" " * R"az"^1
print(rule:match(input) .. " of " .. #input)
```

This will print on the terminal "13 of 12" because all the input has been consumed and the parser stopped at the end of input, which is the 13th "character" in this string. As we can see, the function `rule:match` parses a given input string using a given parser and returns the number of characters parsed. Another way to invoke a parse is using `lpeg.match(rule, input)`, which is equivalent to `rule:match(input)`.

The next example is slightly more complicated. We will parse a comma-separated list of colon-separated key–value pairs.

```
local input = [[foo : bar ,
gorp : smart ,
falcou : "crazy frenchman" ,
name : sam]]
```

The double square brackets denote one of Lua's so-called long strings, which can have embedded newlines. The colons and commas that separate keys and values, and entries, respectively, are surrounded by whitespace. To match all possible optional whitespace we use the set parser and the optional expression.

```
local ws = S" \t\r\n"^0
```

With this, the specification for the key field is one or more letters or digits surrounded by optional whitespace.

```
local name = ws * R("az", "AZ", "09")^1 * ws
```

The value field, on the other hand, can have either the same specification as the key field, which does not allow embedded whitespace, or it can be a quoted string, which allows anything between the quotes. To this end we specify the grammar for a quoted string, which is simply the double quote character, followed by anything that is not a double quote, followed by another double quote. The whole thing may be surrounded by optional whitespace.

```
local quote =
  ws * P'"' *  (1 - P'"')^0 * P'"' * ws
```

Therefore an entry in the key–value list is a `name`, followed by a colon, followed by either a `quote` or a `name`, followed by at most one comma. The whole key–value list may have any number of entries, so we apply the zero or more expression to the aforementioned rule.

```
local keyval =
  (name * P":" * (quote + name) * P","^-1)^0
```

Matching the rule against the input in the same way as the previous example gives "67 of 66".

7 Grammars

The literal parser `P` has a second function. If its argument is a table, the table is processed as a *grammar*. The table has the following layout:

```
P{"<entry point>",
    <non-terminal> = <parsing expression>
    ...
}
```

The string "entry point" is the name of the rule to be processed first. Afterwards the rules are listed in the same manner as they were assigned to variables in the previous example. To refer to non-terminal symbols from within the grammar, the `lpeg.V` function is used. Collecting the aforementioned rules into a grammar could look like this:

```
local rule = P{"keyval",
  keyval =
    (V"name" * P":" * (V"quote" + V"name")
      * P","^-1)^0,
  name =
    V"ws" * R("az", "AZ", "09")^1 * V"ws",
  quote =
    V"ws" * P'"' *  (1 - P'"')^0 * P'"'
      * V"ws",
  ws = S" \t\r\n"^0,
}
```

It becomes a little more verbose because names of non-terminal symbols have to be wrapped in `V"..."`. That is why I personally do not normally include general-purpose rules like the `ws` rule in the example into the grammar, because chances are high I want to use it elsewhere again. The level of verbosity might seem like a disadvantage but the encapsulation is much better this way. It also makes it much easier to define recursive rules, as we will see later.

8 Attributes

In the previous section we have parsed some inputs and confirmed their validity by a successful parse, receiving the length of the parsed input. An important question remains: how do we extract information from the input? When a parse is successful, the basic parsers synthesize the value they encountered, which I am going to call their *attributes*. These attributes can be extracted using LPEG's capture operations.

The simplest capture operation is `lpeg.C(patt)` which simply returns the match of `patt`. Here we parse a sequence of only lowercase letters and print the result.

```
local rule = C(R"az"^1)
print(rule:match"pizza") -- pizza
```

Another, very powerful, capture is the table capture `lpeg.Ct(patt)` which returns a table with all captures from `patt`. This allows us to write a simple parser for comma-separated values (CSV) in only three lines:

```
local cell = C((1 - P"," - P"\n")^0)
local row = Ct(cell * (P"," * cell)^0)
local csv = Ct(row * (P"\n" * row)^0)

local t = csv:match[[
```

```
a,b,c
d,e,f
g,,h]]
```

The variable `t` now holds the table representing the CSV file and we can access the elements by `t[<row>][<column>]`, e.g. to access the “e” in the middle of the table we can use `t[2][2]`.

There are two more captures we need to see, the grouping capture and the folding capture. The grouping capture `lpeg.Cg(patt [, name])` groups the values produced by `patt`, optionally tagged with `name`. The grouping capture is mostly used in conjunction with the folding capture `lpeg.Cf(patt, func)` which folds the captures from `patt` with the function `func`. The most common application is parsing of key–value lists. The key and the value are captured independently at first but are then grouped together. Finally they are folded together with an empty table capture.

```
local key = C(R"az"^1)
local val = C(R"09"^1)

local kv = Cg(key * P":" * val) * P","^-1
local kvlist = Cf(Ct"" * kv^0, rawset)

kvlist:match"foo:1,bar:2"
```

9 More useful parsers

Now that we know how to parse input and extract data, we can start constructing parsers that are more useful. We will next write a parser for floating point numbers. The parser presented here has some limitations. It doesn't handle an integer part that only contains a sign, i.e. `-.1` will not parse. It also doesn't handle hexadecimal, octal, or binary literals. (Consider these to be left as exercises to the reader.)

With these limitations in mind, let's take a look at what floating point numbers look like:

$$\underbrace{\overbrace{\texttt{+123}}^{\text{integer part}}\texttt{.}\overbrace{\texttt{45678}}^{\text{fractional part}}}_{\text{mantissa}}\underbrace{\texttt{e-90}}_{\text{exponent}}$$

With that we formulate the first rule in our grammar, namely

```
number = (V"int" * V"frac"^-1 * V"exp"^-1)
         / tonumber,
```

i.e. a number has an integer part, followed by an optional fractional part, followed by an optional exponent. The apparent division by `tonumber` that we see here is called a *semantic action*. A semantic action is applied to the result of the parser *ad-hoc*. In general it is a bad idea to use semantic actions, because they don't fit into the concept of recursive parsing and introduce additional state to keep track of. Nevertheless there are some cases when semantic actions are useful, as in this case, where we know that what we just parsed is a number and we merely convert the resulting string into Lua's number type.

Now let's parse the integer part. Here I show all the rules that go into it at once.

```
int = V"sign"^-1 * (R"19" * V"digits"
                               + V"digit"),
sign = S"+-",
digit = R"09",
digits = V"digit" * V"digits" + V"digit",
```

So the integer part is an optional sign, followed by a number between 1 and 9, followed by more digits or just a single digit. A sign is one of the characters `+` or `-`. A single digit is just a number between 0 and 9. The `digits` rule is recursive, because many digits are either a single digit followed by more digits, or just that single digit.

Next is the fractional part, which is straightforward. It is just a period followed by digits.

```
frac = P"." * V"digits",
```

Last, the exponential part, which is also relatively simple. It is either a lower- or uppercase E, followed by an optional sign, followed by digits.

```
exp = S"eE" * V"sign"^-1 * V"digits",
```

Now let's check this parser with some test input. We expect the result to be the same number that we input and we expect it to be of Lua type `number`.

```
local x = number:match("+123.45678e-90")
print(x .. " " .. type(x))
```

Output: 1.2345678e-88 number

The full code of the number parser is included in the JSON parser in the Appendix.

10 Complex data formats: JSON

JSON is short for JavaScript Object Notation and is a lightweight data format that is easy to read and write for both humans and machines. JSON knows six different data types of which two are collections. These are `null`, `bool`, `string`, `number`, `array`, and `object`. This maps nicely to Lua where `null` maps to `nil`, `bool` maps to `boolean`, `string` and `number` map to their like-named counterparts, and `array` and `object` both map to Lua's `table` type.

Finally, on the top level there is always an object. Here's an example JSON file [5]:

```
{"menu": {
  "id": "file",
```

```
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New",
       "onclick": "CreateNewDoc()"},
      {"value": "Open",
       "onclick": "OpenDoc()"},
      {"value": "Close",
       "onclick": "CloseDoc()"}
    ]
  }
}}
```

Before we begin writing a parser for this, let's introduce a few general purpose parsers first, which are also not part of the grammar.

```
local ws = S" \t\n\r"^0
```

This rule matches zero or more whitespace characters, where whitespace characters are space, tab, newline and carriage return.

```
local lit = function(str)
    return ws * P(str) * ws
end
```

This function returns a rule that matches a literal string surrounded by optional whitespace. This is useful to match keywords.

```
local attr = function(str,attr)
    return ws * P(str) / function()
            return attr
        end * ws
end
```

This function returns an extension of the previous rule, in that it matches a literal string and if it matched returns an attribute using a semantic action. This is very useful for parsing a string but returning something unrelated, e.g. the `null` value of JSON will be represented by Lua's `nil`.

As mentioned before, at the top level a JSON file expects an object, so this will be the entry point:

```
local json = P{"object",
```

As discussed before, JSON supports different kinds of values, so we want to map these in our parsing grammar.

```
value =
    V"null_value" +
    V"bool_value" +
    V"string_value" +
    V"number_value" +
    V"array" +
    V"object",
```

So, a `value` is any of the value types defined by the JSON format. Now we have to define what these values are and how to parse them. We begin with the easiest ones, the `null` and `bool` values:

```
null_value = attr("null", nil),
bool_value = attr("true", true)
           + attr("false", false),
```

These two types are defined entirely by keyword matching. We use the `attr` function to return a suitable Lua value. Next we define how to parse strings:

```
string_value = ws * P'"'
             * C((P'\\"' + 1 - P'"')^0)
             * P'"' * ws,
```

A string may be surrounded by whitespace and is enclosed in double quotes. Inside the double quotes we can use any character that is not the double quote, unless we escape it with a backslash, as in `\"`. The value of the string without surrounding quotes is captured. To parse number values, we will reuse the number parser defined in the previous section

```
number_value = ws * number * ws,
```

This concludes the parsing of all the simple data types. We move on to the aggregate types, starting with the array.

```
array = lit"["
      * Ct((V"value" * lit","^-1)^0)
      * lit"]",
```

An array is thus a comma-separated list of values, enclosed in square brackets. The list is captured as a Lua table. The final and most complicated type to parse is the object:

```
member_pair = Cg(V"string_value" * lit":"
                 * V"value") * lit","^-1,
object = lit"{"
       * Cf(Ct"" * V"member_pair"^0, rawset)
       * lit"}"
```

An object is a comma-separated list of key–value pairs enclosed in curly braces, where a key–value pair is a string, followed by a colon, followed by a value. To pack this into a Lua table, we use the grouping and folding captures mentioned above. This concludes the JSON grammar.

```
}
```

The full code of the parser is given in the Appendix with a little nicer formatting. Now we can go ahead and parse JSON files.

```
local result = json:match(input)
```

The variable `result` will hold a Lua table which can be indexed in a natural way. For example, if we had parsed the JSON example given in the beginning of this section, we could use

```
print(result.menu.popup.menuitem[2].onclick)
-- OpenDoc()
```

In this way, we could write configuration files for our document, parse them on-the-fly when firing up LuaTEX, and configure the style and content according to the specifications.

11 Summary and outlook

Parsing even complex data formats like JSON is relatively easy using LPEG. A possible next step would be to parse the LuaTEX input file in the `process_input_buffer` callback and replace templates in the file with values from JSON.

Acknowledgements

I'd like to thank the TUG bursary for funding, which supported me in attending this conference.

References

[1] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars. *Software: Practice and Experience* **39**(3), 221–258 (2009).

[2] T. Hoekwater, LuaTEX. *TUGboat* **28**(3), 312–313 (2007). `tug.org/TUGboat/tb28-3/tb90hoekwater-luatex.pdf`

[3] M. Caisse, Using Spirit X3 to Write Parsers. CppCon 2015. `youtube.com/watch?v=xSBWklPLRv`

[4] Wikipedia, Parsing expression grammar. `wikipedia.org/wiki/Parsing_expression_grammar`

[5] D. Crockford, JSON Example. `json.org/example.html`

Appendix: Full code listing of JSON parser

```
local lpeg = require"lpeg"
local C, Cf, Cg, Ct, P, R, S, V =
  lpeg.C, lpeg.Cf, lpeg.Cg, lpeg.Ct, lpeg.P,
  lpeg.R, lpeg.S, lpeg.V

-- number parsing
local number = P{"number",
  number = (V"int" * V"frac"^-1 * V"exp"^-1)
           / tonumber,
  int = V"sign"^-1 * (R"19" * V"digits"
                      + V"digit"),
  sign = S"+-",
  digit = R"09",
  digits = V"digit" * V"digits" + V"digit",
  frac = P"." * V"digits",
  exp = S"eE" * V"sign"^-1 * V"digits",
}

-- optional whitespace
local ws = S" \t\n\r"^0

-- match literal string surrounded by whitespace
local lit = function(str)
  return ws * P(str) * ws
end

-- match literal string and synthesize
-- an attribute
local attr = function(str,attr)
  return ws * P(str) /
              function() return attr end * ws
end

-- JSON grammar
local json = P{
  "object",

  value =
      V"null_value" +
      V"bool_value" +
      V"string_value" +
      V"number_value" +
      V"array" +
      V"object",

  null_value =
      attr("null", nil),

  bool_value =
      attr("true", true) + attr("false", false),

  string_value =
      ws * P'"' * C((P'\\"' + 1 - P'"')^0)
         * P'"' * ws,

  number_value =
      ws * number * ws,

  array =
      lit"[" * Ct((V"value" * lit","^-1)^0)
      * lit"]",

  member_pair =
      Cg(V"string_value" * lit":" * V"value")
      * lit","^-1,

  object =
      lit"{"
      * Cf(Ct"" * V"member_pair"^0, rawset)
      * lit"}"
}
```

◇ Henri Menke
9016 Dunedin
New Zealand
henrimenke (at) gmail dot com

Design into 3D: A system for customizable project designs

William Adams

Abstract

Design into 3D is a system for modeling parametric projects for manufacture using CNC machines. It documents using OpenSCAD to allow a user to instantly see a 3D rendering of the result of adjusting a parameter in the Customizer interface, saving parameters as JSON files which are then read into a LuaLaTeX file which creates a PDF as a cut list/setup sheet/assembly instructions and uses MetaPost to create SVG files which may be loaded into a CAM tool. A further possibility is using a tool such as TPL (Tool Path Language) to make files which are ready to cut.

1 iTeX

It has been almost ten years since Prof. Knuth made the earthshaking announcement of iTeX (see fig. 1; my thanks to Robin Laakso, executive director who kept track of her keepsake as I did not). For the folks who were not fortunate enough to be able to attend: `youtube.com/watch?v=eKaI78K_rgA` (from `tug.org/tug2010/program.html`).

The announcement posited a successor to TeX which would among other things, support 3D, and output to:

- lasercutters
- embroidering machines
- 3D printers
- plasma cutters

all of which are examples of Computer Numeric Control (CNC) machines. Presumably other machines such as mills and routers would also have been supported. While 3D printers have a straightforward mechanism for creating parts (load a 3D file into a "slicing" application), and laser and plasma cutters are limited to 2D (with the possibility of repeated passes for lasers), mills and routers afford the limitation of a 2.5D movement of the tool over and around the part, and the flexibility of using tooling with different shapes which will allow efficient cutting of surfaces with finishes not readily achieved with other tools. They also afford the possibility of loading stock larger than the working area and either cutting it incrementally (known as tiling) or cutting only a small portion of the stock (e.g., when cutting joinery into the end of a board).

Figure 1: iTeX keepsake

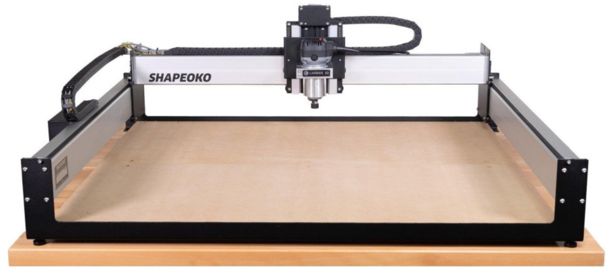


Figure 2: Shapeoko 3 (XXL)

2 CNC machines

Since then, CNC machines have become far more affordable and accessible, mostly due to the open sourcing of the Enhanced Machine Controller,[1] and the development of Grbl which runs on the inexpensive Arduino,[2] with one early machine on its third iteration[3] (see fig. 2).

I happened to pick up a Shapeoko 1 (an open source hobbyist CNC machine based on Bart Dring's MakerSlide,[4] which uses the open source G-Code interpreter Grbl running on an Arduino) early on, and became involved in the project doing documentation and so forth, and now work for the company as off-site tech support.

3 CAD/CAM

CNC is driven by Computer Aided Design (CAD), and Computer Aided Manufacturing (CAM). Most applications thus far developed for this follow the same basic concept: Draw a design or shape, select elements of it and assign appropriate toolpaths to those elements. This works, but can be tedious and repetitive, especially when a design needs some

[1] `www.nist.gov/publications/use-open-source-distribution-machine-tool-controller`

[2] `bengler.no/grbl`

[3] `carbide3d.com/shapeoko`

[4] `www.kickstarter.com/projects/93832939/makerslide-open-source-linear-bearing-system`

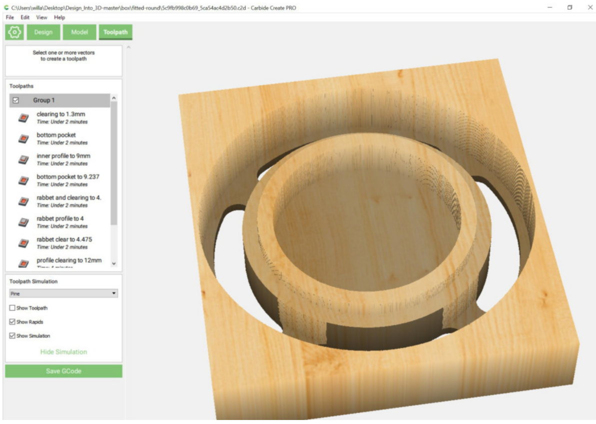


Figure 3: Carbide Create user interface

adjustment such as size or the inclusion or removal of a feature.

Even a simple, small, round box scrolls off the interface when enumerating all of its toolpath settings, making it tedious to transfer said settings to a different project, let alone set them up in the first place. (Fig. 3 shows Carbide Create,[5] a freely available CAD/CAM program).

Some 3D CAM tools do afford options for exporting settings and loading them into different projects, but then one is restricted to the toolpaths which a 3D CAM tool can create, and must work up a 3D model of the project in question. While the latter may not be much of a limitation, the former certainly is.

4 Tagging vs. parameters

TeX works from the idea of a manuscript, assigning to it macros/tagging/markup which then allow the text to be typeset. Moreover, (LA)TeX typically doesn't describe a document as fully as would be needed to make it into a finished object, omitting considerations such as signatures, binding method, and usually the design of a physical cover or dust jacket. Unfortunately, the CAD/CAM workflow doesn't allow for the sort of free-flowing narrative which even a rigorous scientific paper would allow. For example, it may be possible to define a potential project concisely:

- Project type: Box
- Shape: Round
- Lid style: Fitted
- Number of compartments: 1
- Box dimensions:
  - Diameter 50.8mm
  - Height 16.175mm

[5] carbide3d.com/carbidecreate

but there are no readily accessible tools for taking such specifications (or parameters) and directly and immediately creating the design in a format a computer can work with. Parametric tools do allow one to create such designs, but the design has to be created (or programmed) in such a tool.

5 Parametric CAD

That last does indicate a class of tool which is suited for this sort of work: Parametric CAD applications allow one to use numbers, formulae, and algorithms directly to define a design. Commercial examples:

- Autodesk Fusion 360/Inventor
- CATIA V5
- NX
- Onshape
- Pro Engineer
- Rhino 3D (when using the Grasshopper plug-in)
- Solid Edge
- SolidWorks

Many open source applications have also been developed which afford this style of design:

- FreeCAD — unfortunately somewhat limited in the calculations which may be performed; using a spreadsheet is advocated as a work-around[6], and importing OpenSCAD files is also an option.
- NaroCAD
- OpenVSP (Vehicle Sketch Pad from NASA)
- SolveSpace — fully graphical, with parameter alteration requiring selection.
- Varkon

But the most notable implementations are those which are programmatic in nature. Arguably there are too many to name (especially as any programming language can be one), but of special note are:

- Antimony — regrettably available for only GNU/Linux and Mac OS X; previous versions were the subject of the developer, Matt Keeter's, academic thesis.
- Maker.JS — a Microsoft Garage Project, this tool supports 2D design, but requires special effort to create a 3D file or preview.
- OpenSCAD — the most popular tool, widely used for 3D printing, and is notable for support on the popular project-sharing site Thingiverse which inaugurated the "Customizer" feature.
- PLaSM
- Tool Path Language (TPL) — a relatively recent development, this is a JavaScript variant supporting creation of G-Code to control the machine.

[6] floatingcam.com/blog/freecad-parametric-design

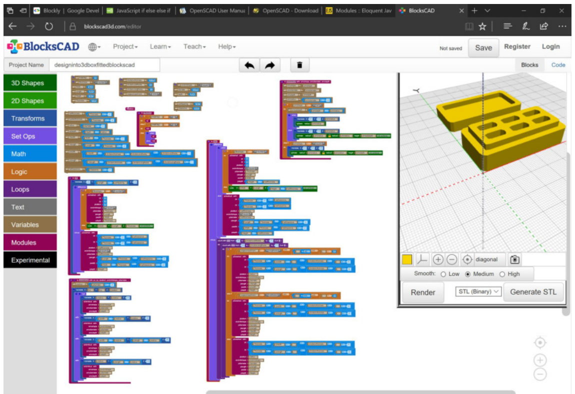


Figure 4: Blockly interface and graphical code of the initial prototype

This attempt at a representative sampling includes the most popular implementation, OpenSCAD, which was used for the implementation of this project.

6 BlocksCAD

Initial development was done using the Blockly implementation of OpenSCAD BlocksCAD (see fig. 4).[7] There are a number of similar tools, with varying tradeoffs, compromises, and difficulties. A better tool, with better graphical integration (specifically, the ability to select nodes, edges, or faces and drag them) would make for even easier development.

BlocksCAD allows one to save a project as an XML file, and to export to OpenSCAD. Similar tools include OpenJSCAD and Flood Editor.

7 OpenSCAD

BlocksCAD allowed a rapid development without worrying about the trivialities of coding such as the placement of semi-colons and an easy conversion into the textual OpenSCAD.

More important for the project is where the Customizer features (unfortunately unsupported by BlocksCAD) were implemented; see fig. 5.

8 Presets

Once a design has been worked up using the customization interface, the parameters must be passed to other tools. Fortunately, OpenSCAD implements saving design settings as “presets” in a JSON file:

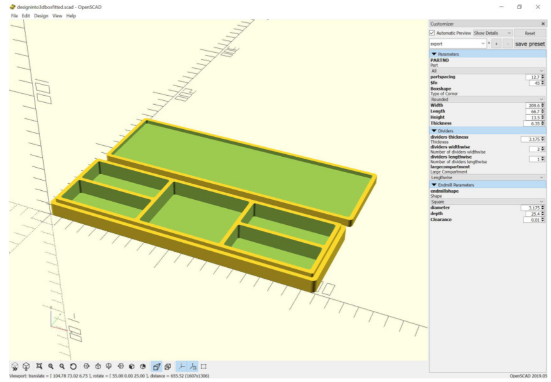


Figure 5: OpenSCAD design with Customizer


```
{
    "parameterSets": {
        "export": {
            "$fn": "45",
            "Boxshape": "0",
            "Clearance": "0.01",
            "Height": "13.5",
            "Length": "66.675",
            "PARTNO": "0",
            "Thickness": "6.35",
            "Width": "209.55",
            "depth": "25.4",
            "diameter": "3.175",
            "dividers_lengthwise": "1",
            "dividers_thickness": "3.175",
            "dividers_widthwise": "2",
            "endmillshape": "1",
            "largecompartment": "2",
            "partspacing": "12.7"
        }
    },
    "fileFormatVersion": "1"
}
```


It is then a matter of loading the JSON data into variables. The first tool which makes use of this is LuaLaTeX, as well as the embedded METAPOST interpreter. Fortunately, the Lua scripting language has a tool available for importing JSON data.[8] Also, Henri Menke (at the conference) demonstrated an elegant system for reading in JSON which merits investigation (see pp. 129–135 in this issue).

```
\newcommand{\boxspecification}{export}
%\typein[\boxspecification]{What preset to use?}
\begin{luacode}
function read(file)
    local handler = io.open(file, "rb")
    local content = handler:read("*all")
    handler:close()
    return content
end
JSON = (loadfile "JSON.lua")()
local table = JSON:decode(read(
                    "designinto3dboxfitted.json"))
```

[7] www.blockscad3d.com/editor

[8] regex.info/blog/lua/json

First, define a macro for each value which may then be redefined at need:

```
%          "PARTNO": "0",
\newcommand{\PARTNO}{\relax}
\newcommand{\definePARTNO}[1]
  {\renewcommand{\PARTNO}{#1}}
```

Then read in each variable from the selected preset (in this case, assigned to the LaTeX macro `\boxspecification`):

```
PARTNO = (table['parameterSets']
                ['\boxspecification']['PARTNO'])
```

Define the contents of the matching TeX macro:

```
\definePARTNO{\directlua{tex.print(PARTNO)}}
```

9 Drawing

Once one has all the numbers loaded, it's a matter of defining macros (the actual path definitions are quite lengthy):

```
def rp (expr x,y,z,w,l,t,d) = draw ⟨outer path⟩;
enddef;
def rpf (expr x,y,z,w,l,t,d,f) =
  fill ⟨inner block⟩ cycle withgreyscale f;
enddef;
def rpu (expr x,y,z,w,l,t,d) =
  unfill ⟨boundary⟩ -- cycle; enddef;
```

and then using them to draw:

```
beginfig(1);
rpf(-diam,-diam,0, Width*u+diam*2, Length*u
    +diam*2, Thickness*u, diameter*u,0.0);
rpu(0,0,0,Width*u, Length*u, Thickness*u, diam);
rpf(Thickness/2*u-halfclearance*u, Thickness/2*u
    -halfclearance*u, 0, Width*u-Thickness*u
    +Clearance*u, Length*u-Thickness*u+Clearance*u,
    Thickness *u-Thickness/4*u, diam,0.5);
endfig;
```

and to fill in the project description:

```
\sbox{\projectdescription}{\vtop{PARTNO:
                                  \dltw{PARTNO}\par
Boxshape: \dltw{Boxshape}\par
Clearance: \dltw{Clearance}\par
Height: \dltw{Height}\par
Length: \dltw{Length}\par
Thickness: \dltw{Thickness}\par
Width: \dltw{Width}\par
depth: \dltw{depth}\par
diameter: \dltw{diameter}\par
dividers:\par
\quad lengthwise: \dltw{dividerslengthwise}\par
\quad thickness: \dltw{dividersthickness}\par
\quad widthwise: \dltw{dividerswidthwise}\par
endmillshape: \dltw{cuttershape}\par
largecompartment: \dltw{largecompartment}\par
partspacing: \dltw{partspacing}}}%
```

to end up with what's shown in fig. 6.

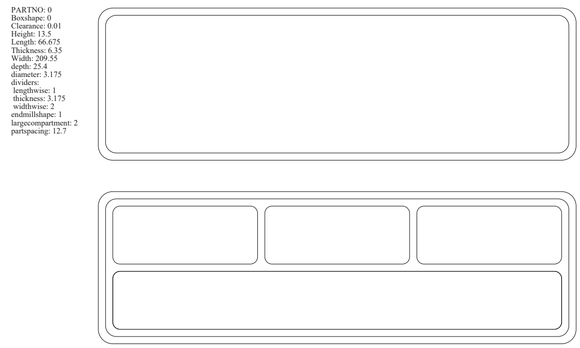


Figure 6: Typeset project plans and parameters

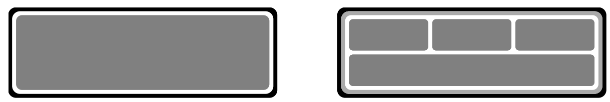


Figure 7: MetaPost output

This is a basic representation — it would be possible to elaborate on that, adding colour and a depth mapping notation, and identify the parts (in this case, lid and base), and for more complex instructions, generate assembly instructions. It is also possible to add geometry and colour code such images so that they can be cut out directly.

The system includes code for making SVG files which may be directly imported into a CAM tool.

```
outputtemplate := "%j-%c.svg";
prologues := 3;
outputformat := "svg";
input designinto3dboxfittedpreamble;
input designinto3dboxfittedfigure1;
input designinto3dboxfittedfigure2;
input designinto3dboxfittedpostamble;
```

The preamble and postamble files have macros and code for cleaning things up. The final drawings are shown in fig. 7.

Creating SVG files allows one to use METAPOST only on the drawings, which is quick and efficient, and to use an SVG viewer (here, nomacs from Image Lounge (`nomacs.org`), shown in fig. 8) to interactively edit and remake the files, adjusting until things are as desired.

Once the files were ready, they could be imported into a CAM tool (in this case, the free Carbide Create) and toolpaths assigned so as to prepare the

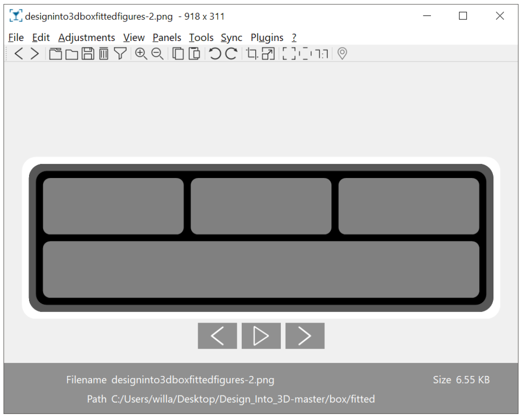


Figure 8: nomacs interactive interface

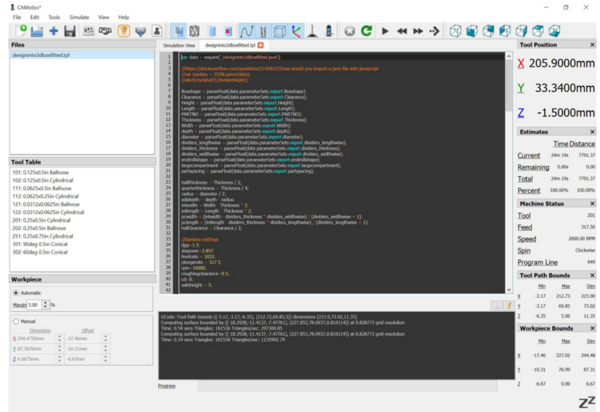


Figure 10: Code for making toolpaths

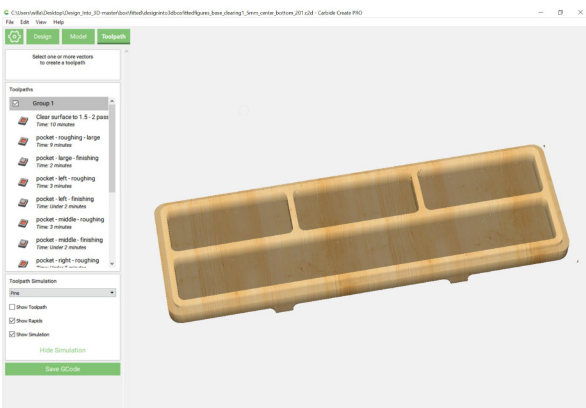


Figure 9: Toolpaths in Carbide Create

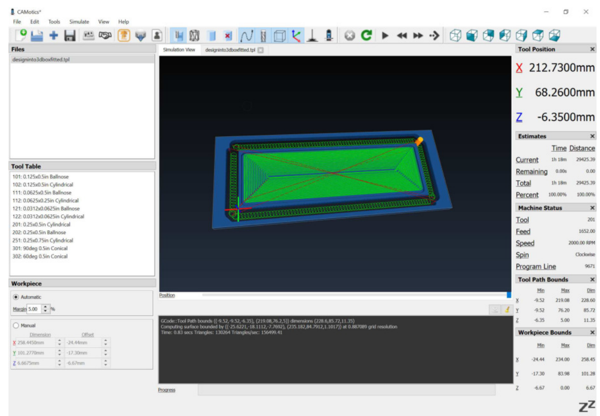


Figure 11: Preview of toolpaths

project for cutting, as shown in fig. 9. This however, limits one to the capabilities of the program in question, and requires a fair bit of manual effort.

10 Coding

The normal output for toolpaths is G-Code (RS-274), developed by the Electronic Industries Alliance in the early 1960s. For lack of a CAM tool which will directly map such vector greyscale images to efficient toolpaths we have instead chosen to work up a program based on CAMotics (`camotics.org`) which will import the JSON data parameters and directly create the toolpaths which will allow the design to be cut out, shown in fig. 10. The results of running this are in fig. 11.

In addition to reading in the parameters, ideally this tool would create optimal toolpaths using advanced features such as:

- ramping in — moving into a cut on a diagonal, or in a helical motion, rather than a straight vertical plunge — endmills are four times better at side-to-side cutting than they are at drilling.
- trochoidal toolpaths — shown in the curlicue paths around the perimeter of the part in fig. 11, trochoidal toolpaths allow efficient removal of material in a narrow slot by reducing tooling engagement, avoiding the full engagement of the machine attempting to move directly through material which has not yet been cut away.
- adaptive clearing — similar to trochoidal toolpaths, this is an optimized motion to clear an area, minimizing redundant motion while keeping tooling engagement at or near optimum.
- roughing clearance and finishing passes — the best finish and most precise/accurate parts are achieved by allowing the machine to remove a minimal amount of material at the end of a cut — much of the complexity shown in the toolpaths shown above were the result of manually implementing these.

11 Cutting

Once toolpaths are created, whether programmatically or using a typical CAM tool, the project may then be cut on the machine (fig. 12).

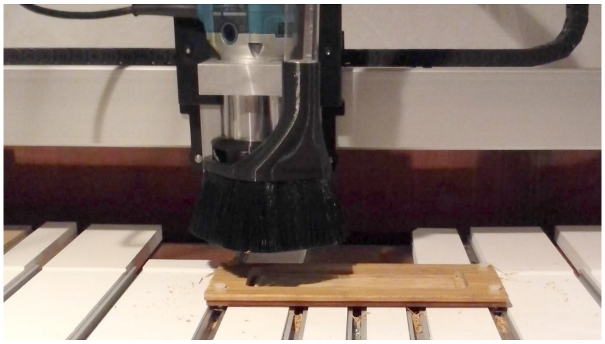


Figure 12: Project cutting

Figure 13: Post-processing after the cut

Once cut, the part will usually require some sort of post-processing (fig. 13) — at a minimum sanding, but possibly cutting tabs to release it from surrounding stock, or cutting away material at the bottom of the profile which was not completely removed.

But once, post-processed, one has a completed project (fig. 14).

12 Concepts

The Tool Path Language program proves the concept of beginning-to-end automation, but raises further questions, and leaves much room for improvement:

- Each project design must be worked up as a collection of specific programs — is there some way to have a more general design language which allows a more natural description of designs?
- The OpenSCAD customization interface is quite limited — an early version attempted to implement a natural switching between Imperial and metric units, but this was so awkward that it was abandoned — using another tool to develop a front-end would seem better.
- It requires that the user download, install, and use a number of tools (OpenSCAD, LuaLaTeX, CAMotics/Tool Path Language) — this on top of the normal program(s) required to run the machine.

Figure 14: Completed project

12.1 Shapes

The above code, rather simplistically, only requires clearing rounded corner pockets. More complex projects will require macros/functions for additional shapes, and names for them. Arranging them by the number of points, we find that all but a few have an accepted single word nomenclature (or suitably concise description):

- 0
  - circle
  - ellipse (requires some sort of non-arc curve)
    - egg-shaped (oval)
  - annulus (one circle within another, forming a ring)
  - superellipse (see astroid below)
- 1
  - cone with rounded end (arc) — see also “sector” under 3 below
- 2
  - semicircle/circular/half-circle segment (arc and a straight line); see also sector below
  - arch — curve possibly smoothly joining a pair of straight lines with a flat bottom
  - lens/vesica piscis (two convex curves)
  - lune/crescent (one convex, one concave curve)
  - heart (two curves)
  - tomoe (comma shape) — non-arc curves
- 3
  - triangle
    - equilateral
    - isosceles
    - right triangle
    - scalene
  - (circular) sector (two straight edges, one convex arc)
    - quadrant (90°)
    - sextants (60°)
    - octants (45°)

  - deltoid curve (three concave arcs)
  - Reuleaux triangle (three convex arcs)
  - arbelos (one convex, two concave arcs)
  - two straight edges, one concave arc
    * An example is the hyperbolic sector[9]
  - two convex, one concave arc
- 4
  - rectangle (including square)
  - parallelogram
  - rhombus
  - trapezoid/trapezium
  - kite
  - ring/annulus segment (straight line, concave arc, straight line, convex arc)
  - astroid (four concave arcs)
  - salinon (four semicircles)
  - three straight lines and one concave arc

Is the list of shapes for which there are not widely known names interesting for its lack of notoriety?

- two straight edges, one concave arc — oddly, an asymmetric form (hyperbolic sector) has a name, but not the symmetrical — while the colloquial/prosaic "arrowhead" was considered, it was rejected as being better applied to the shape below. (It's also the shape used for the spaceship in the game Asteroids (or Hyperspace), but that is potentially confusing with astroid.) At the conference, Prof. Knuth suggested "dart" as a suitable term.
- two convex, one concave arc — with the above named, the term "arrowhead" is freed up to use as the name for this shape.
- three straight lines and one concave arc.

The first in particular is sorely needed for this project (it's the result of inscribing a circle in a square or other regular geometric shape). Do these shapes have names in any other languages which might be used instead?

A final consideration: It has been said that there are two types of furniture — the system fails to take that into account or to leverage on it.

12.2 Two types of furniture

What are the two types of furniture?

- Boxes
- Platforms

This first project has involved making two-piece boxes out of solid materials, simply removing what is not needed for the design. While this works for small pieces, it is necessarily limited to the degree to which it can be scaled up, and quickly becomes profligately wasteful of material. The traditional solution for this is joinery, of which there are many sorts, and thus far for CNC, usually involve complicated fixtures and jigs and multiple setups.

12.3 Further steps

Developing a solution which could incorporate joinery efficiently is one obvious next step. All of the pockets assume 2.5D cutting on a single plane — the ability to make cuts at an angle would afford a welcome flexibility which is needed in some sorts of joinery. Similarly, the ability to make cuts using arbitrary endmill shapes may enable designs as yet undreamed of. Possibilities:

- Joinery
- General purpose design frameworks/grammars
- Special purpose tools — there are many extant project generators for various sorts of boxes, furniture (chairs and workbenches) gears, geography, clocks, even houses — other possibilities include telescopes, cribbage boards, &c.
- Would it be possible to create a font where a series of letters would describe discrete aspects of a design, assign toolpaths to the appropriate letters using that font, and then to change the design by just changing the text?
- Ornamentation — that's next year's Kickstarter and presentation — ideas include Sheridan (traditional Western floral leatherworking), Celtic knots and letters, and Arabesques, as well as various arrangements of text.

13 Continuing work

This was initially a (funded) Kickstarter.[10] It is being developed as a wiki page on the Shapeoko project[11] with code on GitHub.[12] A number of sample files and projects have already been made[13,14,15]; and this is tied into a Thingiverse project[16] and an online box generator.[17]

⋄ William Adams
`willadams (at) aol dot com`

[9] `en.wikipedia.org/wiki/Hyperbolic_sector` and `www.reddit.com/r/Geometry/comments/bkbzgh/is_there_a_name_for_a_3_pointed_figure_with_two`

[10] `kickstarter.com/projects/designinto3d/design-into-3d-a-book-of-customizable-project-desi`
[11] `wiki.shapeoko.com/index.php/Design_into_3D`
[12] `github.com/WillAdams/Design_Into_3D`
[13] `cutrocket.com/p/5c9fb998c0b69`
[14] `cutrocket.com/p/5cb536396c281`
[15] `cutrocket.com/p/5cba77918bb4b`
[16] `www.thingiverse.com/thing:3575705`
[17] `chaunax.github.io/projects/twhl-box/twhl.html`

The design of the HINT file format

Martin Ruckert

Abstract

The HINT file format is intended as a replacement of the DVI or PDF file format for on-screen reading of TeX output. Its design should therefore meet the following requirements: reflow of text to fill a window of variable size, convenient navigating of text with links in addition to paging forward and backward, efficient rendering on mobile devices, simple generation from existing TeX input files, and an exact match of traditional TeX output if the window size matches TeX's paper size.

This paper describes the key elements of the design and motivates the design decisions.

Why do we need a new file format?

The first true output file format for TeX was the DVI format [3]. When PostScript became available, it was soon supplemented by `dvips` [7], and now, most people I know use `pdftex` to produce TeX output in PDF format. There are two good reasons for that: partly, the PDF format is a near-perfect match [4] for the demands of the TeX typesetting engine, but first and foremost, the PDF format is in widespread use. It enables us to send documents produced with TeX to practically anybody around the globe and be sure that the receiver will be able to open the document and that it will print exactly as intended by its author (unless a font is neither embedded in the file nor available on the target device) .

But the main limitation of the PDF format is its inherent inability to adapt to the given window size. For reading documents on mobile devices, the HTML format is a much more convenient format. Part of the concept of HTML is a separation of content and presentation: the author prepares the content, the browser decides on the presentation — at least in principle. It turns out that designers of web pages spare no effort to control the presentation, but often the results are poor. Different browsers have different ideas about presentation, users' preferences and operating systems interfere with font selection, and all that might conflict with the presentation the author had in mind.

When it comes to ebooks, the popular epub format [2] is derived from HTML and inherits its advantages as well as its shortcomings. As a consequence, ebooks when compared with printed books are often of inferior quality.

What is needed is a document format which meets the demands of the TeX typesetting engine and that gives the author as much control over the presentation as possible but still can adapt to a given paper format — be it real or electronic paper. Building on previous work [8, 9], these two design objectives guided the development of the HINT file format.

While the TeX typesetting engine, its internal representation of data, its algorithms, and its debugging output, was the driving force of the development of the HINT file format, giving the whole project its name (the recursive acronym for "HINT Is Not TeX"), the result is not limited to the TeX universe. In the contrary, it makes the best parts of TeX available to any system that uses the HINT file format.

Faithful recording of TeX output

At the beginning of the design, the primary necessity was the ability to faithfully capture the output of the TeX typesetting engine.

To build pages, TeX adds nodes to the so-called "contribution list". The content of a HINT file is basically a list of all these nodes, from which a viewer can reconstruct the contributions and build pages using TeX's original algorithms. So with few exceptions, TeX nodes are matched one-to-one by HINT nodes.

Of course, we need characters, ligatures, kerns, rules, hlists and vlists; and as in TeX, dimensions are expressed as scaled points. But even a simple and common construction like `\hbox to \hsize {...}` requires new types of nodes: this is a horizontal list that may contain glue nodes and has a width that depends on `\hsize` which is not known when the HINT file is generated. To express dimensions that depend on `\hsize` and `\vsize`, HINT uses linear functions $w + h \cdot \texttt{\hsize} + v \cdot \texttt{\vsize}$, called *extended dimensions*. Linear functions are a good compromise between expressiveness and simplicity. The computations that most TeX programs perform with `\hsize` and `\vsize` are linear and in the viewer, where `\hsize` and `\vsize` are finally known, extended dimensions are easily converted to ordinary dimensions. Necessarily, HINT adopts TeX's concepts of stretchability, shrinkability, glue, and leaders.

One of the highlights of TeX is its line breaking algorithm. And because line breaking depends on `\hsize`, it must be performed in the viewer. But wait — an expensive part of line breaking is hyphenation and this can be done without knowledge of `\hsize`. So HINT defines a paragraph node, its width is an extended dimension, and all the words in it contain all possible hyphenation points in the form of TEX's discretionary hyphens. To maintain complete compatibility between TEX and HINT, two types of hyphenation points had to be introduced: explicit and automatic. TEX uses a three pass approach for breaking lines: In the first pass, TEX does not attempt automatic hyphenation and uses only discretionary hyphens provided by the author. Likewise HINT will use in its first pass only the explicit hyphenation points. Given the same value of `\hsize`, TEX and HINT will produce exactly the same line breaks. In a paragraph node, HINT also allows vadjust nodes and a new node type for displayed formulas to make sure that the positioning of displayed equations and their equation numbers is exactly as in TEX.

The present HINT format also has an experimental image node that can stretch and shrink like a glue node. Therefore, images stretch or shrink together with the surrounding glue to fill the enclosing box. The insertion of images in TEX documents is common practice. But TEX treats images as "extensions" that are not standardized. In a final version of HINT, I expect to have a more general media node. I think it is better to have a clearly defined, limited set of media types that is supported in all implementations than a wide variation of types with only partial support.

One node type of TEX that is not present in HINT is the mark node. TEX's mark nodes contain token lists, the "machine code" for the TEX interpreter, and for reasons explained next, HINT does not implement token lists.

Efficient and reliable rendering

On mobile devices, rendering must be efficient and files must be self-contained. To meet these goals, the proper foundation is laid in the design of the file format.

The most important decision was to ban the TEX interpreter from the rendering application. A HINT file is pure data. As a consequence, TEX's output routines (and with them mark nodes) were replaced by a template mechanism. Templates, while not as powerful as programs, will always terminate and can be processed efficiently. Whether they offer sufficient flexibility remains to be seen. It is a fact, however, that very few users of TEX or LATEX write their own output routines. So it can be expected that a collection of good templates will serve most authors well.

The current template mechanism of HINT is still experimental. It is sufficient to replace the output routines of plain TEX and LATEX.

HINT files contain all necessary resources, notably fonts and images, making them completely self-contained. Embedding fonts makes HINT files larger — the effect is more pronounced for short texts and less significant for large books — but it makes HINT files independent of local resources and of local character encodings. Indeed, a HINT file does not encode characters, it encodes glyphs. While HINT files use the UTF-8 encoding scheme, it is possible to assign arbitrary numbers to the glyphs as long as the assignment in the font matches the assignment in the text. The only reason not to depart from the standard UTF-8 encoding is to maximize compatibility with other software, e.g., to search for user-entered strings or for text to speech translation.

Zoom and size changes

On mobile devices it is quite common to switch within one application between landscape or portrait mode to use the screen space as efficiently as possible. Further, users usually can adjust the size of displayed content by zooming in or out.

For rendering a HINT file, these operations simply translate into a change of `hsize` and `vsize`, with consequences for line and page breaking. While changing line breaks affects only individual paragraphs, changing a page break has global implications, making precomputing page breaks impractical. Consequently, the HINT file format must support rendering either the next page or the previous page based solely on the top or bottom position of the current page. In turn, this implies that it must be possible to parse the content of a HINT file in both forward and backward directions.

A HINT file encodes TEX's contribution list in its content section. To support bidirectional parsing, each encoding of a node starts with a tag byte and ends with that same tag byte. From the tag byte, the layout of the encoding can be derived. So decoding in the backward direction is as simple as decoding in the forward direction.

Changes in TEX's parameters, for example paragraph indentation or baseline spacing, pose another problem for bidirectional parsing. HINT solves this problem by using a stateless encoding of content. All parameters are assigned a permanent default value. To specify these defaults, HINT files have a definition section. Any content node that needs a deviation from the default values must specify the new values locally. To make local changes efficient, nodes in the content section can reference suitable predefined lists of parameter values specified again in the definition section, described next.

Simple and compact representation

At the top level, a HINT file is a sequence of sections. To locate each section in the file, the first section of a HINT file is the directory section: a sequence of entries which specify the location and size of each section. The first entry in the directory section, the root entry, describes the directory section itself. The HINT file format supports compressed sections according to the `zlib` specification [1]. Using the directory, access to any section is possible without reading the entire file.

The directory section is preceded by a banner line: It starts with the four byte word `hint` and the version number; it ends with a line-feed character. The directory section is followed by two mandatory sections: the definition section and the content section. All further sections, containing fonts, images, or any other data, are optional. The size of a section must be less than or equal to 2^{32} bytes. This restriction is strictly necessary only for the content section. It sets a limit of about 500 000 pages and ensures that positions inside the content section can be expressed as 32-bit numbers.

For debugging, the specification of a HINT file also describes a “long” file format. This long file format is a pure ASCII format designed to be as readable as possible. Two programs, `stretch` and `shrink`, convert the short format to the long format and back. They are literate programs [5], and constitute the format specification [10].

Since large parts of a typical content section contain mostly character sequences, there is a special node type, called a text node, optimized for the representation of plain text. It breaks with two conventions that otherwise are true for any other node: The content of a text node cannot be parsed in the backward direction, and it depends on a state variable, the current font. To mitigate the requirement for forward parsing, the size of a text node is stored right before the final tag byte. This enables a parser to move from the final tag byte directly to the beginning of the text. Since text nodes cannot span multiple paragraphs, they are usually short.

Inside a text node, all UTF-8 codes in the range $2^5 + 1$ to 2^{20} encode a character in the current font; codes from `0x00` to `0x20` and `0xF8` to `0xFF` are used as control codes. Some of these are reserved as shorthand notation for frequent nodes. For example, the space character `0x20` encodes the interword glue, and others introduce font changes or mark the start of a node given in its regular encoding.

The two forms of content encoding, as regular nodes or inside a text node, introduce a new requirement: when decoding starts at a given position, it must be possible to decide whether to decode a regular node, a UTF-8 code, or a control code. Control codes have only a limited range and the values of tag bytes can be chosen to avoid that range. Conflicts between UTF-8 codes and tag bytes cannot be avoided, hence positions inside text nodes are restricted to control codes. A position of an arbitrary character inside a text node can still be encoded because there is a control code to encode characters (with a small overhead).

Clear syntax and semantics

Today, there are many good formal methods to specify a file format, and the time when file formats were implicit in the programs that would read or write these files seems like ancient history. The specification of the HINT file format, however, is given as two literate programs: `stretch` and `shrink`. The first reads a HINT file and translates it to the “long” format and the second goes the opposite direction and writes a HINT file.

Of course, these programs use modern means such as regular expressions and grammar rules to describe input and output and are, to a large extent, generated from the formal description using `lex` and `yacc`. For this purpose, the `cweb` system [6] for literate programming had to be extended to generate and typeset `lex` and `yacc` files. I consider this representation an experiment. I tried to combine the advantages of a formal syntax specification with the less formal exposition of programs that illustrate the reading and writing process and can serve as reference implementations. The programs `stretch` and `shrink` can also be used to verify that HINT files conform to the format specification.

Specifying semantics is a difficult task and a formal specification is entirely impossible if the correctness depends partly on personal taste. Fortunately the new file format is just an “intermediate” format as part of the TeX universe. So the following commutative diagram is an approximation to a formal specification.

$$\begin{tikzcd} \texttt{file.tex} \arrow[r, "\text{HiTeX}"] \arrow[dr, "\text{TeX}"'] & \texttt{file.hnt} \arrow[d, "\textsf{HINTcl}"] \\ & \texttt{file.dvi} \end{tikzcd}$$

The programs HiTeX and HINTcl mentioned in the diagram are currently in development. HiTeX is a modified version of TeX that produces HINT files as output; HINTcl is a command line program which reproduces TeX's page descriptions as if the parameter `\tracingoutput` were enabled. While it does not actually produce a DVI file, its output can be compared to the page descriptions in TeX's `.log` file to make sure the diagram above would indeed be commutative. The prototypes available so far do not yet support all the features of TeX or HINT.

Conclusion

The experimental HINT file format proves that file formats supporting efficient, high quality rendering of TeX output on electronic paper of variable size are possible. The upcoming prototypes for a TeX version (HiTeX) that produces such files and viewer programs on Windows and Android will provide a test environment to investigate and improve concepts and performance in practice.

In the long run, I hope that a new standard for electronic documents will emerge that enjoys widespread use, has the output quality of real books, is easy to use and powerful enough to encode TeX output, offers the author maximum control over the presentation of her or his work, and can cope with the variations in screen size and screen resolution of modern mobile devices.

References

[1] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC Editor, 1996.
`tools.ietf.org/html/rfc1950`

[2] EPUB 3 Community Group. epub 3.
`w3.org/publishing/groups/epub3-cg`

[3] D. Fuchs. The format of TeX's DVI files. *TUGboat* 3(2):14–19, Oct. 1982.
`tug.org/TUGboat/tb03-2/tb06software.pdf`

[4] H. Hagen. Beyond the bounds of paper and within the bounds of screens; the perfect match of TeX and Acrobat. In *Proceedings of the Ninth European TeX Conference*, vol. 15a of *MAPS*, pp. 181–196. Elsevier Science, Sept. 1995.
`ntg.nl/maps/15a/09.pdf`

[5] D. E. Knuth. *Literate Programming.* CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[6] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation.* Addison Wesley, 1994.
`ctan.org/pkg/cweb`

[7] T. Rokicki. *Dvips: A DVI-to-PostScript translator.* `tug.org/dvips`

[8] M. Ruckert. Computer Modern Roman fonts for ebooks. *TUGboat* 37(3):277–280, 2017.
`tug.org/TUGboat/tb37-3/tb117ruckert.pdf`

[9] M. Ruckert. HINT: Reflowing TeX output. *TUGboat* 39(3):217–223, 2019. `tug.org/TUGboat/tb39-3/tb123ruckert-hint.pdf`

[10] M. Ruckert. *HINT: The File Format.* Aug. 2019. ISBN 978-1079481594.

◇ Martin Ruckert
Hochschule München
Lothstrasse 64
80336 München
Germany
`ruckert (at) cs dot hm dot edu`

TEXFolio — a framework to typeset XML documents using TEX

Rishikesan Nair T., Rajagopal C.V., Radhakrishnan C.V.

Abstract

TEXFolio is a web-based framework on the cloud to generate standards-compliant, hyperlinked, bookmarked PDF output directly from XML sources with heavy math content, using TEX. TEXFolio is a complete journal production system as well. It can produce strip-ins which are alternate GIF or SVG images of MathML content. In addition, DOI look-up, HTML rendering of XML/MathML, and the whole dataset generation according to the customer's specification are also possible. Customer-specific validation tools can be integrated in this system.

1 Introduction

TEXFolio is a web-based complete journal production system that accepts XML documents as input and generates a variety of outputs per user directives. At the moment, TEXFolio accepts documents tagged per the NLM/JATS or Elsevier Journal Article DTD. The typesetting engine is TEX, which allows hyperlinked, bookmarked and standards-compliant PDF outputs of infinite variants in terms of look and feel. It further permits TEX authors to directly edit their documents at the proof stage and a master copy editor can pass it for publishing with minimal loss of time.

Although the underlying engine of TEXFolio hasn't deviated from the genre of free/libre software, the computing paradigms have markedly shifted to those in vogue to make the system modernized and competitive in terms of usability, technologies and performance. In fact, TEXFolio allows users to undertake any stage of work anywhere in the world, owing to its absolute compatibility with cloud and mobile computing. That is further augmented by the usage of LATEX3 methodologies and programming to perfect the production system to an efficient, automated and accurate one.

2 The workflow

2.1 XML first

The input can be either XML or LATEX. XML must be per NLM/JATS DTDs or the Elsevier Journal Article DTD. TEXFolio ingests these two types of XML documents and generates a LATEX file by applying corresponding an XSLT stylesheet over the document. This LATEX file is used to edit the content and/or to generate PDF output.

The processing cycle of XML → LATEX → edit → PDF can be repeated any number of times, as shown in the schematic diagram provided in Figure 1.

Whenever the LATEX file is edited, the user can either generate PDF or go through another cycle of XML → LATEX → PDF to ensure high fidelity between XML and PDF, thereby making it a truly XML-first workflow. We call it the "XROUND" process.

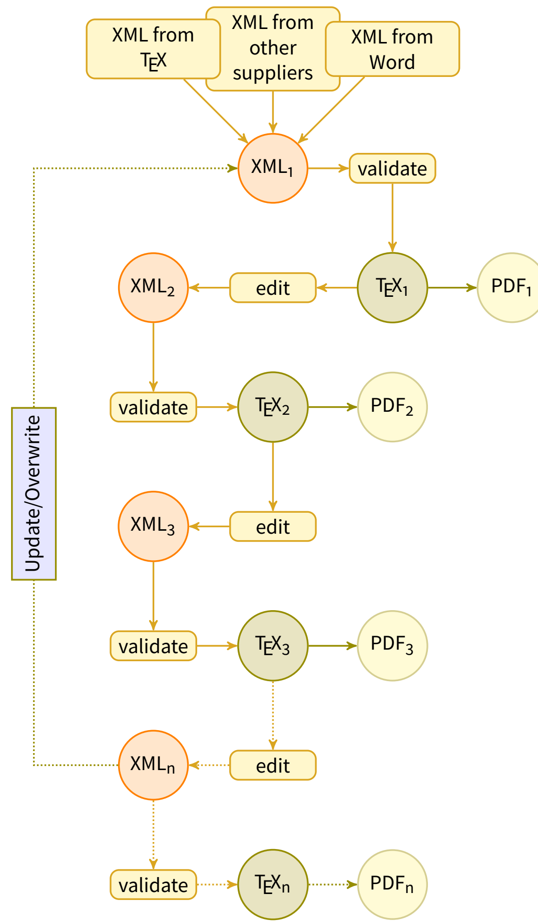


Figure 1: XML-first workflow

2.2 LATEX input

For LATEX input, we need to restructure the document to augment XML generation since the front matter is very verbose and structured in a granular way in XML documents. The bibliography must be provided as a BIBTEX database. Barring the front and back matters, the main body of the document does not pose any problem during XML generation. The system can digest all the author's macro definitions, `newtheorem`-type declarations, or any other declarations provided by the AMS math packages and other standard LATEX packages.

TEX4ht is our preferred engine to translate LATEX documents to XML format among all the tools and available software around. Since TEX is being used to process the document, it can easily and effectively handle all the macros and user definitions or declarations seamlessly. See Figure 2 for a schematic diagram of the workflow.

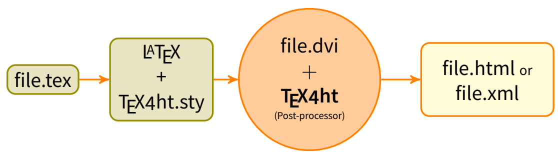


Figure 2: LATEX to XML generation diagram.

3 Features

A list of the most commonly used features built into TEXFolio are given here.

3.1 General

Cloud With the development of TEXFolio, we are able to do text processing in the cloud rather than on a personal computer. This is critically important in the current scenario. Previously, text processing software was installed on each user's desktop and updating the systems with the most recent changes always caused problems. In the text processing world, much software is subject to regular updates and ensuring these updates was a herculean task. With the deployment of TEXFolio, we need to update the software on the server only and users' desktops need not be touched.

TEX For the whole process (i.e., generate XML, PDF, strip-ins, etc.), we are using TEX and friends. TEX being the most sophisticated software for high-quality typesetting, TEXFolio ensures high-quality output especially for mathematics, computer science, economics, engineering, linguistics, physics, statistics.

Low learning curve With the help of user-friendly and an attractive interface, even a novice user can use it without much learning. Since standard LATEX commands are used, a normal LATEX user can quickly start using it.

Cross-platform Since it is browser-based, users with different operating systems can access it without any difficulty.

Browser, sole software A desktop with any current browser and an Internet connection is all that is required to access TEXFolio. Instead of a desktop machine, if you have a Raspberry Pi, that is more than enough.

Self-publishing Supporting self-publishing is another feature of TEXFolio. TEXFolio serves well the "author as publisher" since it accepts TEX documents as input and can easily generate PDF, HTML 5, NLM/JATS or Elsevier Journal Article XML outputs if the sources are marked up per the `elsarticle` or `stm` document classes. These are the deliverables required for a web platform. As mentioned above, the bibliography needs to be provided as a BIBTEX database.

3.2 Inputs

TEXFolio can currently accept the following input formats. The first is for a LATEX-first workflow whereas the second and third are for an XML-first workflow.

LATEX In the case of a LATEX-first workflow, the input source has to be structured according to `neptune.cls`, `elsarticle.cls` or `stm.cls`.

NLM/JATS XML The user loads the XML file in one of these DTDs. TEXFolio generates a TEX file from this loaded JATSXML. From now on, the source will be this machine-generated TEX file. Using this, the user can paginate, make changes and/or generate different deliverables from this source.

Elsevier Journal Article XML The user loads an Elsevier DTD XML file. TEXFolio generates a TEX file from the input. Just as with NLM/JATS, from now on, the source will be the machine-generated TEX file, and the user can generate different deliverables from this source.

3.3 Outputs

Whatever the input (i.e., LATEX or XML), the user can generate the following output files from the source in a few seconds.

PDF The PDF output generated will be according to the standards. You may generate a web version PDF as well as a print version PDF. The Web version will be hyperlinked, bookmarked and according to the PDF/A-1 standards whereas the print ready or fat PDF will be according to PDF/X-1a standards. It can easily be configured by a developer if the user wants a PDF according to another ISO standard.

HTML5 + MathML This is another deliverable or output which can be generated from the LATEX or XML workflow. MathJax is supported.

XML The user can always generate a client version XML file in either the LATEX-first or XML-first workflow. The XML used for generating PDF can have processing instructions embedded if any vital instructions for TEX was lost in the

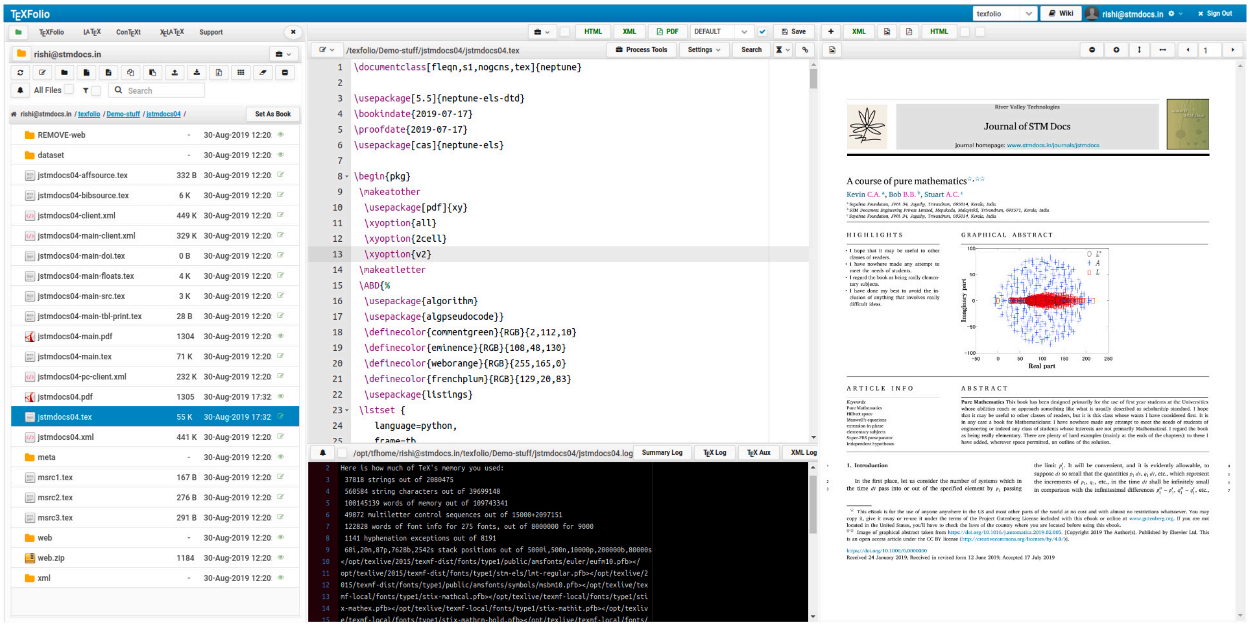


Figure 3: Main page of TEXFolio: File manager (left), text editor (center), output viewer (right), log windows (bottom), tools (top).

translation to XML. Before delivering to the client, these processing instructions can also be removed, and any other necessary changes made. Some publishers even require the XML document to be on a single physical line. Currently, it can ship either NLM/JATS XML + MathML or Elsevier Journal Article XML + MathML.

e-pub It is not one of the standard outputs at present, but is easily configurable.

MathML Math in the above XML documents will be tagged as MathML, SVG, GIF/PNG/JPG, TEX math, or all. Both MathML 2.0 and 3.0 are supported.

3.4 More features

1. Depends on LATEX3 methodologies and paradigms.
2. DOI link fetching and checking
3. Crossmark, ORCID, FundRef linking
4. Linking external objects such as Genbank accession numbers, PDB, CTGOV, OMIM, etc.
5. Source editing with track-changed source, PDF and XML at proof/final stages
6. Author proofing with Neptune
7. Tooltips in PDF
8. Technical support for TEX authors
9. NLM validator to check against NLM Article Publishing, CrossRef Deposit Schema and PMC style checker
10. Supports pdfLATEX, ConTEXt and XƎLATEX for PDF creation. However, for standard-compliant PDF generation, pdfLATEX is used for the time being.

4 Summary

Figure 3 shows the main page of TEXFolio.

For more details and screenshots, please visit `https://texfolio.org`.

◇ Rishikesan Nair T., Rajagopal C.V., Radhakrishnan C.V.
STM Document Engineering Pvt. Ltd., River Valley Campus, Mepukada, Malayinkil, Trivandrum 695571, India
`rishi (at) stmdocs.in`
`http://stmdocs.com`

NEPTUNE — a proofing framework for LaTeX authors

Aravind Rajendran, Rishikesan Nair T., Rajagopal C.V.

Abstract

NEPTUNE is a web-based proofing framework for LaTeX authors. It is part of TeXFolio, the complete journal production system in the cloud.

NEPTUNE accepts author-submitted LaTeX documents (with or without enrichment and restructuring) as well as machine-generated LaTeX documents from XML sources. Authors can edit LaTeX sources as in any standard editor with additional features.

Starting from the end of November 2018 when NEPTUNE was first released, the framework has been used for author proofing of more than 2,500 articles in more than 100 journals, through August 31, 2019.

1 Introduction

In academic publishing, LaTeX authors may be considered difficult, since they insist on better typography, adherence to conventions (particularly in math equations), and use of their finely crafted LaTeX sources for final output by utilizing myriad benefits offered by LaTeX. In recent times, galley proofs are provided to authors as editable sources as a web page in XML or HTML format. Authors who have submitted their articles in LaTeX format often dislike viewing and editing their output on a web page since the original LaTeX sources for math is not provided. Further, embedded Ti*k*Z graphics, XY-pic and commutative diagrams, `prooftree` math, and the like are replaced with their respective graphics, denying any opportunity to edit in case of mistakes. Source code with packages like `listings` suffers a similar fate … the woes are many. Hence, LaTeX authors are not without cause when they complain of publishers' lack of typographic and semantic sensibilities.

Neptune is an answer for all these problems, wherein a LaTeX author can be provided with copy-edited LaTeX sources and corresponding PDF output in the final print format side by side with enough facilities to navigate between source and PDF, a navigable list of track changes showing copy edits that can be accepted or rejected, a navigable list of author edits made during the proofing session, comparison of pre- and post-proof LaTeX sources side by side with the ability to discard any edit, comparison of pre- and post-edit PDF versions, navigable query lists, multiple sessions for proofing, standard editor features, etc.

2 Where to start?

The typesetter uploads the author's proof to Neptune and sends the link to the author. Clicking the link will take the author to the opening page of Neptune where instructions are given. A [Proceed] button enables the author to access the LaTeX source and PDF output of the proof. The general interface is shown in Fig. 1.

The author can edit the LaTeX source and confirm changes in the PDF after recompiling (the menu bar has a [Compile] button).

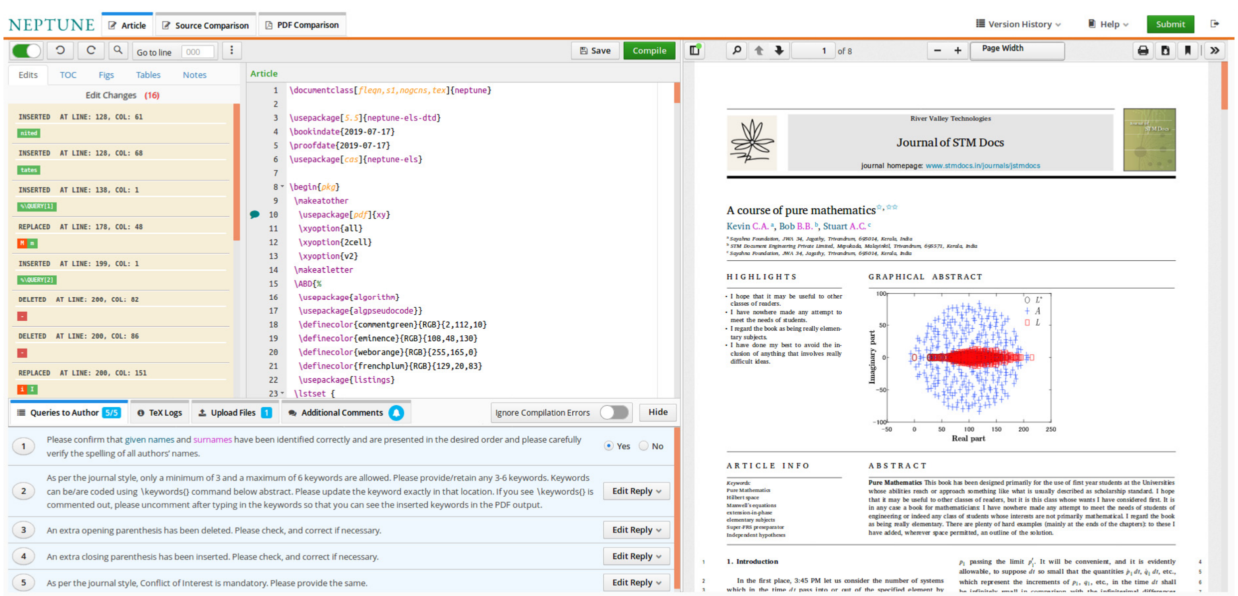


Figure 1: Neptune — Main page.

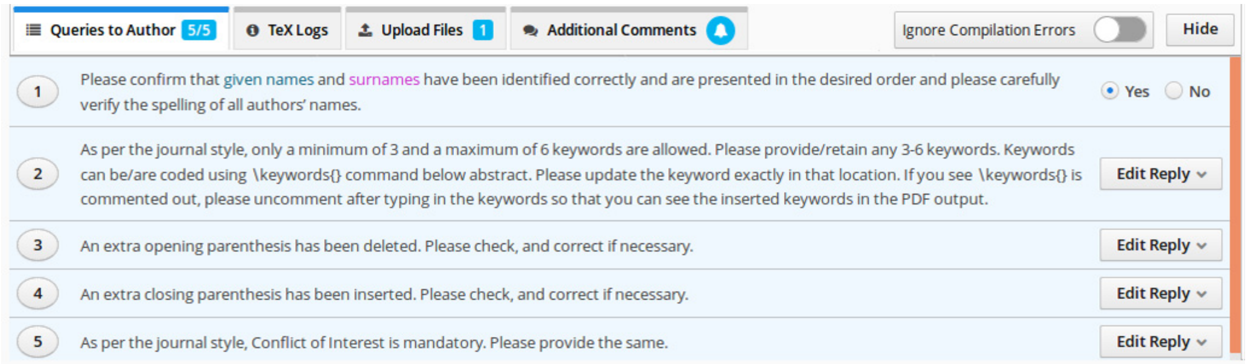


Figure 2: Query window and Ignore compilation error feature.

3 The process

As a web application, Neptune provides facilities to edit LaTeX documents as with any desktop text editor. While keeping the native LaTeX experience, several other additional features have been provided to make the job easier.

Neptune allows editing text in any area of the document and adding or removing any object (section level headings, figures, tables, math, list items, cross references, citations, bibliography items, ...). If the editing results in any counter changes, all objects will be re-numbered and cross-references and citations will be fixed automatically.

The PDF output can be generated any time and can be downloaded if needed.

4 General editing

There is nothing special to say about general editing of text. The usual text attributes: bold (`\textbf`), italics (`\emph`, `\textit`); font attributes like sans serif (`\textsf`), fixed width font (`\texttt`), small caps (`\textsc`); size changing commands (`\large`, `\small`, `\footnotesize`); and so forth all work as one would expect.

Moreover, you may insert sections, paragraphs, floats such as figures, tables, etc., inline or display math equations, theorems and similar environments, bibliographic items, cross-references, etc.

In short, all standard commands in general text manipulations work fine without any surprises.

5 Main features

In addition to the general editing features, other main features are listed below:

5.1 Article, Source Comparison and PDF Comparison tabs

The three main tabs are Article, Source Comparison, and PDF Comparison. The Article tab contains mainly features for editing, compiling, functional tracker, resolving queries, seeing TeX logs, upload files, PDF viewer, versioning control, etc. See Figs. 1 and 2.

The Source Comparison tab is for comparing the copy-edited source (provided to the author as the source of a galley proof) with the author-edited source. Using this facility, authors can compare the two TeX sources and verify the changes. Synchronised movement of both TeX files is available, with a scroll button to move both TeX files simultaneously, which helps make the comparison easier.

Similar to the Source Comparison tab, the PDF Comparison tab is for comparing copy-edited PDF file (again provided to the author as a galley proof) with author-edited PDF file. Synchronised movement of both PDFs is enabled in this tab also.

5.2 Synchronized pre/post-edited sources

Pre- and post-edited document sources, along with a tracker window with hyperlinked list of edit changes, are available. Authors can make last minute checks and confirm all edits or discard any change at will.

5.3 Source–PDF navigation

One-to-one links between the source TeX file to PDF and back are available, making it easier to navigate from source to the corresponding location in the PDF and vice-versa. The user needs to compile the sources once for this feature to take effect.

5.4 Notes, requests, comments

Any number of notes, requests, comments, etc., can be added to the document sources by clicking at line number. In addition to this, an [Additional Comments] tab is provided to provide a general comment.

5.5 Error-stop/non-stop modes

PDF generation can optionally be stopped at an error or continued until the end of the job, without

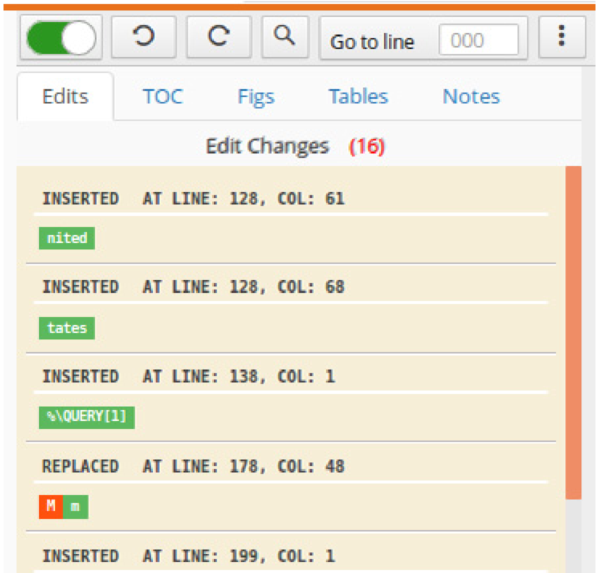


Figure 3: Functional tracker.

stopping at errors, to facilitate an author's preferred style of debugging.

5.6 Functional tracker

A convenient tracker of changes made by a copy editor is available. The line/column numbers of the insertion or deletion are provided. When you click any text in the tracker window, a pop-up with the corresponding item will appear with [Reject] and [Accept] buttons. You can click a button according to your choice. By default, Accept will be applied. See Fig. 3.

5.7 PDF output

At the end of the editing job or at any other time, authors can generate a PDF from their edited sources which is exactly like the one that will be ultimately printed in the journal.

5.8 No need for another proof

Since authors edit directly on the LaTeX sources and view/save the final output as a PDF, there is no need to request (and wait for) a revised proof from the typesetter. This saves considerable production time.

5.9 Version history

Version control systems allow authors to compare files, identify differences, and merge changes if needed prior to committing anything. Neptune's version history facility gives authors full confidence to edit without any fear of losing anything from the source. They are free to save as many versions they want and retrieve any specific version as needed. See Fig. 4.

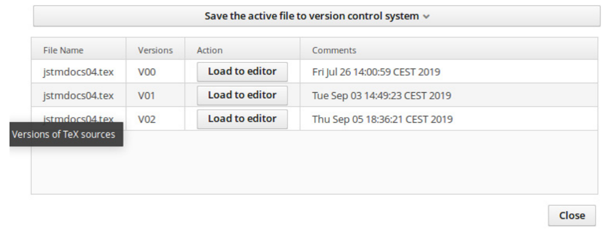


Figure 4: Version control.

5.10 Miscellaneous features

- The PDF output has active hyperlinks and bookmarks.
- Unlimited Undo/Redo is supported.
- Search/replace and regular expressions are supported.
- Neptune works well with Raspberry Pi, thus saving energy, consistent with our environment-friendly production technologies.

6 Supported browsers

Neptune supports the following browsers with version numbers noted against their names or later:

- Firefox: 54+
- Google: Chrome 55+
- Safari: 11.02+
- Internet Explorer: 11+
- Edge 41.16+

7 Success story

Finally the success story.

One of the world's major scientific, technical, and medical publishers recently adopted NEPTUNE as their LaTeX proofing tool. Beginning in November 2018, up through August 31, more than 2500 articles have been proofed through NEPTUNE. The first three months were a pilot period, with only four journals. Continuing to roll out more journals in batches, NEPTUNE now supports more than 100 journals. Before submitting an article, authors can take an optional survey. From this survey, the customer satisfaction score was 95%, showing NEPTUNE as an efficient and user-friendly web proofing framework.

⋄ Aravind Rajendran, Rishikesan Nair T., Rajagopal C.V.
STM Document Engineering Pvt. Ltd., River Valley Campus, Mepukada, Malayinkil, Trivandrum 695571, India
aravind (at) stmdocs.in
http://stmdocs.com

The LaTeX release workflow and the LaTeX `dev` formats

Frank Mittelbach, LaTeX Project

Abstract

How do you prevent creating banana software (i.e., software that gets ripe at the customer site)? By proper testing! But this is anything but easy.

The paper will give an overview of the efforts made by the LaTeX Project Team over the years to provide high-quality software and explains the changes that we have made this summer to improve the situation further.

Contents

Some history

The LaTeX Project team's attempts to provide reliable high-quality software can be traced back to the first days when we took over LaTeX 2.09 from Leslie Lamport for maintenance and started to work on producing a new major version of LaTeX which came into existence around 1993 under the name of LaTeX 2_ε. In its core this is still the version you are using today, albeit these days with many extensions and additional kernel features.

While LaTeX 2_ε looked fairly similar on the user interface level (to allow for easy transition), under the hood it used completely different internal concepts in some parts of the software and was therefore quite different in many areas from the LaTeX 2.09 version. After all, the goal of the new system was to overcome (most) limitations and deficiencies of the original program that had surfaced since its introduction in 1986.

Executing such major software changes was and is a daunting task, especially when a huge user base relies on the software to continue to function seamlessly with old and new documents (and preferably produce identical results).

To have a fighting chance of success, we came up with the idea of a regression test suite for LaTeX where certified results from document runs could be used to automatically check that changes made to the internal code base did not affect the behavior of LaTeX document level interfaces, etc.[1] A good overview of the features and mechanisms of the early regression test suite can be found in [1].

Around 1992 we then initiated several volunteer projects [2], one of which was helping to build a base set of test files which would test all interfaces, and additionally provide unit tests for known bugs and issues that we intended to fix.

It was largely due to Daniel Flipo (coordinator back then) and the volunteers who worked on this initial test set that the introduction of LaTeX 2_ε turned out to be quite successful. Even though we experienced some problems and also some level of push back (as in "Why do we need a new system which is so much bigger and contains all these fonts with accented characters that nobody is ever going to need ..."), on the whole the system was favorably received and after a round of smaller maintenance releases that fixed overlooked issues and added a few more missing bits and pieces, the kernel settled to a stable state with little update or extension. Instead, further development moved into the package area, for which the new kernel provided a better basis than the old.

The growing release support infrastructure

With each bug that was found and any new feature that got added, the test suite grew. These days it contains roughly 700 test documents. There are about 400 for the core of LaTeX and another 300 for the packages that form the required set, i.e., `amsmath`, `graphics` and `tools`, but excluding `babel`, which has its own test suite and release cycle.

At the beginning the support scripts to automatically run the tests were Unix-based (mainly a huge `Makefile` plus a few `bash` scripts). We then switched over to a Windows-based system and finally, when LuaTeX became generally available in the distributions, rewrote the whole system in Lua. Thanks mainly to Joseph Wright, Will Robertson and others this `l3build` program is now a very flexible and sophisticated tool that is not only capable of running the test suite for our own LaTeX work, but also able to automate the whole release cycle up to and including automatic uploads to CTAN [3, 4].

As its actions are configured through a simple but powerful configuration file that you maintain

[1] Nothing especially spectacular these days, but around 1990 it was a rather uncommon approach.

in your package source tree, it is perfectly capable of supporting any sort of package development in the TeX world (it doesn't have to be LaTeX). So if you are a developer and haven't seen it yet, try it out; it will most certainly make your life simpler and through its automation, more reliable.

The typical release workflow for LaTeX

With `l3build`, our typical release work flow these days looks roughly like this:

1. Development phase

- Make some changes (bug fixes or minor extensions) to the LaTeX kernel or to core packages.
- Write some new test files to cover the change and the expected behavior.

2.a Testing phase (run `l3build check`)

- This way we immediately see if the changes break anything.
- All tests are executed with pdfTeX, XeTeX, LuaTeX and for a certain subset also with pTeX and upTeX.
- This means that running `l3build` with the `check` target executes more than 2000 test documents for the LaTeX 2ε distribution, which even on a reasonably fast machine requires some patience.

2.b Testing phase (do a `texmf-dist` search)

- If we had to modify internal kernel commands we do a sweep over the whole TeX Live distribution tree to check if those commands have been used or (often more importantly) modified by other packages and whether or not our change will conflict with that usage.
- If so, we analyze the situation further and inform the package authors to coordinate any necessary updates.
- Depending on the analysis, we may also conclude that we need to revert our change or implement it differently.

2.c Testing phase (asking for user testing)

- This requires manually installing a prerelease of the new kernel locally and thus is unfortunately both somewhat time consuming and requiring knowledge that is not so easy to come by these days.
- As a result we had little to no feedback in the last years from this step.

3. Everything is finally "in the Green"

- … or so everybody believes.

4.a Upload phase (run `l3build ctan`)

- This target reruns all checks with all engines, processes all documentation and if this succeeds without any errors, collects the result in a `.zip` for upload to CTAN.

4.b Upload phase (run `l3build upload`)

- Based on preconfigured information this target automatically uploads the `.zip` file together with the necessary metadata to CTAN. For some pieces of information, such as installation instructions to the CTAN team, we are prompted, as they change from upload to upload.

4.c Upload phase (reaching the distributions)

- Once installed on CTAN the kernel and packages move into the TeX distributions, e.g., TeX Live and MikTeX, and then, depending on the update policy chosen by the end user, show up on the user machines either automatically or through a manually initiated update process. And then …

BOOM!

- Thousands of users use (a.k.a. "test") the changes and a few encounter issues, due to dependencies our test suite hasn't signaled, packages we have overlooked, code or packages that are not on CTAN, etc.

5. Urgent patch phase

- The problem is that with the user base measuring in millions[2] even a rate of 0.001% of users being affected by some issue translates into a noticeable number of users with problems.
- Thus, even a single issue with some nearly-unused package may need urgent correction (and it takes a few days from producing a patch to getting it into end user hands). As a consequence this is usually a phase of hectic activity and we have seen in recent releases more than one patch needing to be provided in quick succession — the worst case was three in 2016.

[2] Nobody knows for sure how many active LaTeX users are out there as there is no easy way to measure this. Downloads of TeX Live or from CTAN, for example, are done through a large network of mirrors and the download numbers per mirror are unknown. But there are somewhat between six and ten thousand hits on the LaTeX project web site per day, most of which look at the "get" or "about" pages, i.e., are most likely new users. Another indication is that visitor numbers grow substantially at the start of university terms. This would mean more than two million prospective new users hitting the site per year.

Reasons for failure

What are the reasons that despite extensive regression testing we often end up with patch releases? They are largely due to the ineffectiveness of steps 2.b and 2.c in the above sequence.

The regression test suite we run in step 2.a ensures that our official interfaces are all working correctly and any bugs we have fixed in the past do not suddenly reappear. In fact on several occasions it has saved us from major blunders by stopping us from distributing "harmless changes that couldn't by any chance produce problems" but did after all — often in, on the surface, unrelated places.

Finding all these dependencies up front

However, even a huge test suite can't find and test for all kinds of possible dependencies in several thousand contributed packages and millions of user documents. This problem is increased by the fact that the LaTeX code was written for a very constrained engine and the kernel is therefore very much tailored and streamlined, saving token space whenever possible.[3] As a result there aren't many interfaces where third-party packages can officially hook into kernel functionality, so it isn't surprising that there is nearly no internal LaTeX command that hasn't been (mis)used in one way or another by some package out there.

This is the reason for the importance of step 2.b, but since this is largely a manual effort it is easy to miss cases or fool oneself into believing that no one could possibly have altered *this or that* internal command in a package — in the end somebody usually did after all.

The missing maintainer problem

The other issue with step 2.b is that these days there are unfortunately many packages in use where the original author is no longer reachable, because he or she has moved on. Their packages are on CTAN and in the distributions but the maintainer information is no longer correct. As long as everything works that is not necessarily a problem, but the moment something breaks it can be quite hard or even impossible to find the person and even if the search is successful it may turn out that they no longer have any interest in their work which they did years ago.

A recent prominent example of this problem is the package `tabu` which implements a nice user syntax on top of `array`, `tabularx` and `longtable`. That package was abandoned by its author around 2011 but people continued to use it despite a few unfixed (minor) bugs, because it does implement a number of nice ideas.

Unfortunately, its code hacks in rather bad ways into the kernel internals in places that should never have been altered by other packages. So when we had to fix a color leakage problem in tabular cells in the core kernel commands by adding color safe groups that then broke the package for good. Without a maintainer who was willing to spend the necessary time to unravel these hacks in the package code, the package remained broken when the 2018 kernel got released.[4]

The biggest problem resulting from this was that `doxygen`, the de-facto open source standard for producing annotated C++ code documentation was making heavy use of the `tabu` package when producing LaTeX-based PDF documents. As a result their toolset was initially unable to use the current LaTeX release. We recently resolved this for them by providing a dedicated rollback of the involved kernel fixes to be used within their workflow (i.e., reintroducing the kernel bug so that a special version of `doxygen-tabu` could be used as part of their documentation tools). This is clearly far from a perfect solution, so we hope that a new maintainer for `tabu` will eventually step forward so that this rollback can be removed again.

No or not enough third-party testing

However, we believe that the most important factor for ending up with patch level releases was in step 2.c: the insufficient public testing of the release prior to its move into the main distributions.

In essence, the effort needed from users was simply too high and the setup too complicated, so only a very small number of people participated. Testing was therefore neither sufficient nor comprehensive.

As a result, overlooked dependencies on third-party packages or failure with typical user input were

[3] In the early '90s when most of this code was initially written, this was an absolute must as TeX's main memory, register space, pool size, etc., was much smaller than today. These days one would produce quite different-looking code that would support extensions much better, by offering the necessary hooks, and this is what we are gradually introducing in various parts of the code. However, given that there are many packages out there that expect the code to look exactly like it does right now, changing anything means that these packages need to have corresponding changes.

[4] In fact the LaTeX Project team tried to update the package when it became clear that nobody was maintaining it, and we managed to produce a version that didn't die right out from the beginning. However, the package altered so many commands and used them in new ways that this emergency fix was only partially successful. So in the end we could only suggest that people should not use it in the future, or more exactly not until a new maintainer stepped forward and spends the necessary time to unravel the coding issues.

seldom found beforehand but only when the release moved to the distributions and everybody became (unwillingly) a tester — banana software after all, despite our best efforts above.

The LaTeX dev formats

To improve this situation and hopefully get to a release workflow that doesn't normally involve step 5, we developed the concept of a LaTeX development format. This format contains a prerelease of the upcoming main LaTeX release and is ready for testing by anybody using either TeX Live or MikTeX.

All the user needs to do is to replace his or her standard engine call by adding the suffix `-dev` to the name, for example, using

```
pdflatex-dev myfile
```

instead of `pdflatex myfile` on the command line. If you use an editing environment with integrated TeX processing, then there is normally some configuration possibility, where you can either make the same change or even add another menu item. Besides pdfTeX, all other major engines are supported as well, e.g., with `lualatex-dev` you get the new format on top of the LuaTeX engine, etc.

Pre-release identification

If you call LaTeX in this way you can immediately see that the pre-release format is used. For example processing this document with the line above gives:

```
This is pdfTeX, Version 3.14... (TeX Live 2019)
(preloaded format=pdflatex-dev)
 restricted \write18 enabled.
entering extended mode
(./TUB-latex-dev.tex
LaTeX2e <2019-10-01> pre-release-2
```

As you can see the format announces itself as a pre-release of the upcoming 2019-10-01 release of LaTeX, and the number tagged at the end indicates that it is the second pre-release we have distributed (the first was a trial to see if the mechanism functions correctly). If bugs are found during the testing (or if we enable further features for the upcoming release) we might issue another pre-release in which case the number would increase accordingly.

However, the important point to note here is that the development format is not like a "nightly build" (that you would get by tracking the LaTeX source at GitHub); rather, it changes only if we think that the code is ready for public testing, i.e., has passed our own internal tests in steps 2.a and 2.b.

How does it work?

The files for the pre-release are uploaded by the LaTeX Project Team to CTAN under the package names `latex-base-dev`, `latex-graphics-dev`, and if necessary `latex-tools-dev`, etc. From there they are integrated into the distributions into the tree `tex/latex-dev/...`, which is not searched by default. Thus, when you are using, say, `pdflatex`, only the files from the main release are used.

However, if any of the programs ending in `-dev` are called, then this extra tree is prepended to the search tree, so that not only the pre-release format is used, but also any other file from that tree, e.g., `article.cls`, is found first. For any package not part of the pre-release, the TeX engine will continue to find it in the main tree and use that version.

This allows any user who works on an important project (such as a thesis or a book) to quickly test if this work continues to typeset correctly under the upcoming format. Similarly, it enables any developer of a package that has known or unknown dependencies on a certain kernel version to check if any adjustments made work well with both the current and upcoming LaTeX release — and if so, upload a new version of his or her work prior to the actual release date of the new LaTeX kernel.

Reporting issues in the dev format

If, during such testing, issues or incompatibilities are found (that in the past would have led to step 5) we suggest that a Github issue is opened for them so that they can be tracked and addressed by the team. Details on how to open such an issue can be found at the LaTeX Project website [5].

References

[1] Frank Mittelbach. A regression test suite for LaTeX 2$_\varepsilon$. *TUGboat*, 18(4):309–311, 1997. `tug.org/TUGboat/tb18-4/tb57mitt.pdf`

[2] Frank Mittelbach, Chris Rowley, and Michael Downes. Volunteer work for the LaTeX3 project. *TUGboat*, 13(4):510–515, 1992. `tug.org/TUGboat/tb13-4/tb37mitt-l3.pdf`

[3] Frank Mittelbach, Will Robertson, and LaTeX3 team. `l3build` – A modern Lua test suite for TeX programming. *TUGboat*, 35(3):287–293, 2014. `tug.org/TUGboat/tb35-3/tb111mitt-l3build.pdf`

[4] Joseph Wright. Automating LaTeX(3) testing. *TUGboat*, 36(3):234–236, 2015. `tug.org/TUGboat/tb36-3/tb114wright.pdf`

[5] LaTeX Project Team. Bugs in LaTeX software. `www.latex-project.org/bugs`

◇ Frank Mittelbach, LaTeX Project
Mainz, Germany
`frank.mittelbach (at) latex-project dot org`
`www.latex-project.org`

Accessibility in the LaTeX kernel — experiments in Tagged PDF

Chris Rowley, Ulrike Fischer, Frank Mittelbach

Abstract

This is a brief summary of a talk given by the first author at the TUG'19 conference, together with some references for further reading and viewing.

1 Introduction

Accessibility requirements for PDF documents are described in two standards: PDF/UA and the more recent PDF2.0. One of the major features they mandate is that the PDF must be properly tagged, so we are investigating how LaTeX can be adapted to easily produce tagged PDF.

The main purpose of this talk was to introduce the experimental package *tagpdf* by the second author. But we start with a quick "bullet points introduction" to the structure of a PDF file and what is meant by "Tagged PDF".

2 PDF in bullets

The first thing to understand about PDF is: In a PDF file (almost) everything is ... *PDF Objects*! Here are two examples of important object types that we always find in a PDF file:

- many objects are Dictionaries, which are simply key-value (property) lists
- other objects are Streams: Text Streams are an important part of each Page Object

Some important particular objects are:

- Resource Objects: containing, for example, font and encoding information
- Page Objects: containing information about a page
- Navigation Objects: these enable quick access to all the important objects within the PDF file.

For Tagged PDF, in addition to Page Objects and their Text Streams, the following are required:

- a Structure Tree Object, whose nodes are PDF Objects (surprise?), with:
- – a root node
- – structure element nodes: each of these is a Dictionary Object containing references to its parent, siblings and child nodes
- – leaf nodes: each containing additional references, each of which is to:
  – a page, plus
  – a "marked part" of that page's Text Stream

The slides for the talk contain examples of the type of code used in a PDF file for defining the objects related to structure and tagging.

3 Philosophy

We believe that the production of documents that exploit the large range of functionality that can nowadays be incorporated into PDF is very important and is fundamental to what LaTeX, as a document processing system, is all about!

We also believe that the production system must pay close attention to the actual, detailed contents of the input LaTeX file: these details must not be ignored as they contain everything that the system knows about the author's intentions.

We are therefore certain that we first need to adapt and enhance the LaTeX kernel to better support all these new ideas. Furthermore we must go on to help package developers and maintainers in exploiting the new possibilities.

We are currently working primarily on getting right the low-level basic coding so that we can build on top of this to add necessary features into the LaTeX system. We do not want to add lots of new stuff on top of the current LaTeX, or to produce a parallel system that will most likely conflict badly with standard LaTeX processing.

4 Current work

Development has started on the experimental package *tagpdf*. Its purposes are summarised here:

- allow experimenters to identify problems they find with tagging, and to discover the support needed for other accessibility requirements;
- develop a code basis for the support of tagging in the LaTeX kernel.

Please note that, being experimental, it needs experimenters to use it, of at least these types:

- authors and users of documents:
  – What is truly needed in an accessible document?
- package maintainers/developers:
  – What is needed for a package to produce accessible output?
  – How can the LaTeX Team help ease the conversion of all packages?

The current (preliminary) version of the *tagpdf* packages provides low-level mark-up commands to support tagging. For example, commands to:

- add structure element nodes to the structure tree
- add "marked content" tags to the content stream

- add to the structure tree nodes all the necessary pointers to the marked content associated with a given node

The package also supports other aspects of accessibility, such as setting up links appropriately, and the input of essential document meta-data. It is well documented with descriptions of how to use it and of how to provide us with feedback. Please do!

The documentation contains more background information about accessibility and tagging, with descriptions of how PDF works and what makes a PDF file accessible. It also lists some currently known problems and how we plan to solve them.

5 Coming soon, we hope!

We of course hope to get many ideas from all you experimenters, but meanwhile we are looking in detail at how the LaTeX kernel can better support tagging and other aspects of accessibility. We are also looking at whether the various TeX engines need any enhancements to better support the production of full-featured PDF.

In the TeX community, Ross Moore and others in TUG's Accessibility Working Group have done considerable work on many of these problems, including the complex subject of how to represent formulas, etc., in accessible PDF. We are therefore actively exchanging ideas with them and we are pleased to thank TUG and DANTE e.V. for their current support of this work.

We are of course also very interested in collaboration with other organisations, individuals and companies who have engineering expertise in this area (from both the TeX perspective and the PDF perspective) and we intend to actively pursue such contacts.

As part of their program to position PDF as a prime source of accessible information in "value-added documents", the expert engineers at the "home of PDF", Adobe, are showing a high level of interest in the use of LaTeX to produce accessible PDF. This gives a clear indication of the importance of LaTeX for the production of PDF documents and we are therefore planning to collaborate closely with them.

References

The *tagpdf* package This is available at:
`https://github.com/u-fischer/tagpdf` and
`https://ctan.org/pkg/tagpdf`.

PDF Standards PDF/UA (PDF/Universal Accessibility) is the informal name for ISO 14289.
On July 28, 2017, ISO 32000-2:2017 (PDF 2.0) was published. See:
`https://www.iso.org/standard/64599.html`
and
`https://www.iso.org/standard/63534.html`

More on PDF Lots of information is available at:
`https://en.wikipedia.org/wiki/PDF`

Moore on PDF Ross Moore has published many talks and articles, see:
`https://maths.mq.edu.au/~ross/TaggedPDF`
Videos of both his talk and this one at TUG 2019 can be seen at:
`http://science.mq.edu.au/~ross/TaggedPDF/TUG2019-movies`

Slides for this talk These are available at:
`https://latex-project.org/publications/indexbyyear/2019/`

TUG's Accessibility Working Group
More information and a fuller bibliography on Accessibility is available at:
`https://tug.org/twg/accessibility/`

◇ Chris Rowley
LaTeX3 Team
chris.rowley (at) latex-project dot org
https://www.latex-project.org

◇ Ulrike Fischer
LaTeX3 Team
fischer (at) troubleshooting-tex dot de
https://www.latex-project.org

◇ Frank Mittelbach
LaTeX3 Team
frank.mittelbach (at) latex-project dot org
https://www.latex-project.org

Creating commented editions with LaTeX — the `commedit` package

Boris Veytsman

1 Introduction

An edition of a classic or sacred text where the original is accompanied by layers of comments is one of the most ancient types of books. One of the most prominent examples is the traditional layout of the Talmud, where the original text is surrounded by the comments (Figure 1). Usually this commented edition is completely different from another subgenre of academic books: a facsimile edition, which faithfully reproduces the original. In the latter case the pages are typeset exactly as in the historical book, and there are no footnote markers in the text, because it would break the integrity of the page. Instead, sometimes endnotes are added, which refer to the page numbers in the main corpus.

One can imagine a combination of facsimile and commented edition: a page of the original is reproduced either in the full size or reduced and typeset together with the notes. Again, Talmud pages (Figure 1) can be an inspiration for this.

Interestingly enough, this style was chosen for a series of students' textbooks accompanied by teachers' materials by Livro Aberto de Matemática, a project of the Institute of Pure and Applied Mathematics, Brazil.[1] According to this design, a teachers' book reproduces students' books, adding to them a layer of comments and discussion as shown in Figure 2. The comments in the teachers' book should appear close to the corresponding page of the students' books. Thus a page of the teachers' book might not be completely filled with the comments; alternatively, in the case of more comments than can be fit around the students' page, the comments may continue on the subsequent pages of the teachers' book.

The package *commedit* [4] is intended to implement this design in LaTeX. The main aim of this package is to create a single source that can be used to produce both students' and teachers' books, or, in other words, the original and the commented versions of the same textbook.

There are many TeX packages that allow one to typeset academic editions, often following EDMAC's ideas [2, 3, 5]. However, none of them allows the easy creation of two books, with and without comments. Thus a new package seemed to be warranted.

[1] `https://impa.br/noticias/projeto-do-impa-propoe-livro-didatico-aberto-e-colaborativo/`

2 User interface

The user interface [4] assumes the main source to be the students' book. It is a conventional LaTeX file with additional text put between `\begin` and `\end` of a special environment. When the students' book is typeset, these environments are omitted from the output. However, they are not completely ignored: LaTeX writes them, along with the information about the page of the students' book where they appeared, into a separate `.tex` file. Typesetting the latter results in the teachers' book.

There are three kinds of special environments. The environment `commeditPreamble` has a mandatory argument *`filename`* — the name of the teachers' book file. It should precede all other special environments and appear only once in the students' book. When LaTeX sees this environment, it opens the file *`filename`*`.tex` and writes to it the preamble using the text until `\end{commeditPreamble}`. The environment `commeditText` contains the chunks of teachers' books which are not tied to pages of students' book: front matter, teachers-only chapters, etc. Lastly, the `commeditComments` environment contains the text that is tied to the pages of the students' book.

There are various parameters and hooks allowing one to customize the way the teachers' book is typeset: paper size, the number of columns, page geometry, etc. See [4] for a detailed user manual.

3 Under the hood

The implementation is based on altering the output routine for both students' and teachers' editions. Below we describe how it is done.

3.1 Students' book

When LaTeX typesets the students' book, it writes the contents of `commeditPreamble` and `commeditText` directly to the teachers' book source, adding some packages and commands to the preamble. However, the contents of `commeditText` environments are written to the teachers' book inside the special `commentsBox` environment.

When LaTeX ships out a page of the students' book, the command `\typesetComments{`*`page`*`}` outputs to the teachers' book, where *`page`* is the "real" (or "PDF") page number, the one which is not reset by the `\pagenumbering` command.

When LaTeX finishes the students' book, it also closes the teachers' book source.

3.2 Teachers' book

When LaTeX typesets the teachers' book, it starts by reading the `.aux` file for the students' book. This

Figure 1: A Talmud page with several layers of comments (typeset in TEX, see [1])

allows one to refer to pages, equations and figures in the students' book using the label-ref system.

The parts of the teachers' book that are not tied to students' book pages are typeset in the usual way. However, when LATEX encounters a `commentsBox` environment (see Section 3.1), it is not typeset. Rather, it is added to a running galley of comments. The command `\typesetComments{page}` triggers the following processes. First, we start a new page of the teachers' book, and put the image of the students' book page on it (we use `\includegraphics` for this). Second, we start to typeset the galley of comments around this image, using `\vsplit` to split it into columns. If the galley is too long to fit on the current page, we start continuation pages, putting the remainder of the galley on them. When the galley is typeset, we clear the page. A typical result is shown in Figure 3.

This method does not allow "real" floats or inserts in the comments — we convert floats to non-floating tables and figures, and footnotes become endnotes.

4 Conclusions

We present a simple way to typeset both commented and original editions from a single source. A combination of students' and teachers' books is an example of the application of this package.

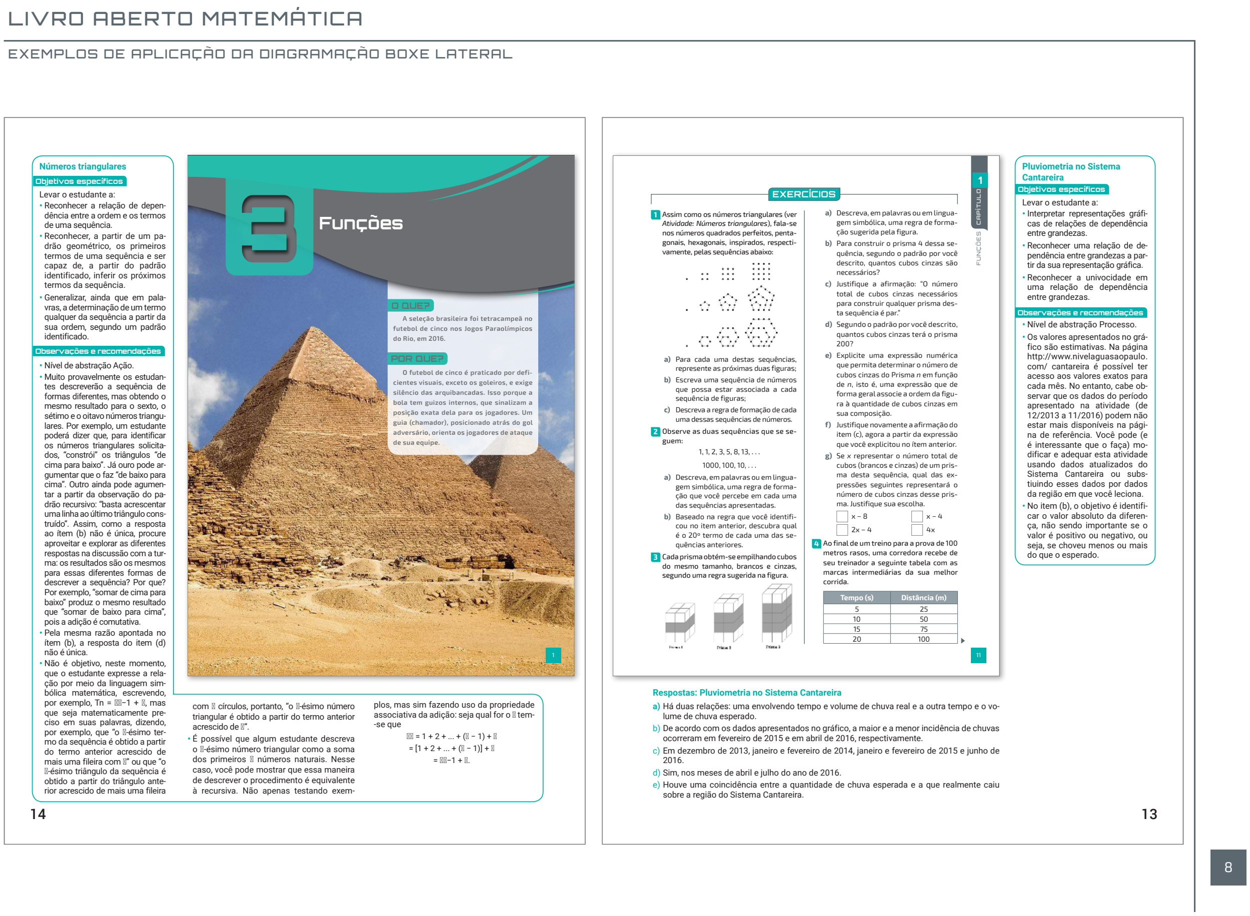


Figure 2: A teachers' book design, Livro Aberto de Matemática

Acknowledgments This work was supported by Livro Aberto de Matemática, Instituto Nacional de Matemática Pura e Aplicada, Brazil. I am grateful to Augusto Quadros Teixeira and Fabio Luiz Borges Simas for their patience.

References

[1] A. Hoenig. *Makor. A System for Typesetting Biblical and Modern Hebrew with Omega and TEX*, 2003. `https://ctan.org/pkg/makor2`

[2] J. Lavagnino and D. Wujastyk. *Critical Edition Typesetting: The EDMAC format for plain TEX*. TEX Users Group and UK TEX Users Group, San Francisco; Birmingham, 1996. `https://ctan.org/pkg/edmac` and `https://tug.org/edmac`

[3] M. Rouquette. *reledmac. Typeset scholarly editions with LATEX*, 2019. `https://ctan.org/pkg/reledmac`

[4] B. Veytsman. *Creating commented editions*, 2019. `https://ctan.org/pkg/commedit`

[5] P. Wilson, H. Press, and M. Rouquette. *ledmac. A presumptuous attempt to port EDMAC, TABMAC and EDSTANZA to LATEX*, 2016. `https://ctan.org/pkg/ledmac`

◇ Boris Veytsman
School of Systems Biology
George Mason University
`borisv (at) lk dot net`

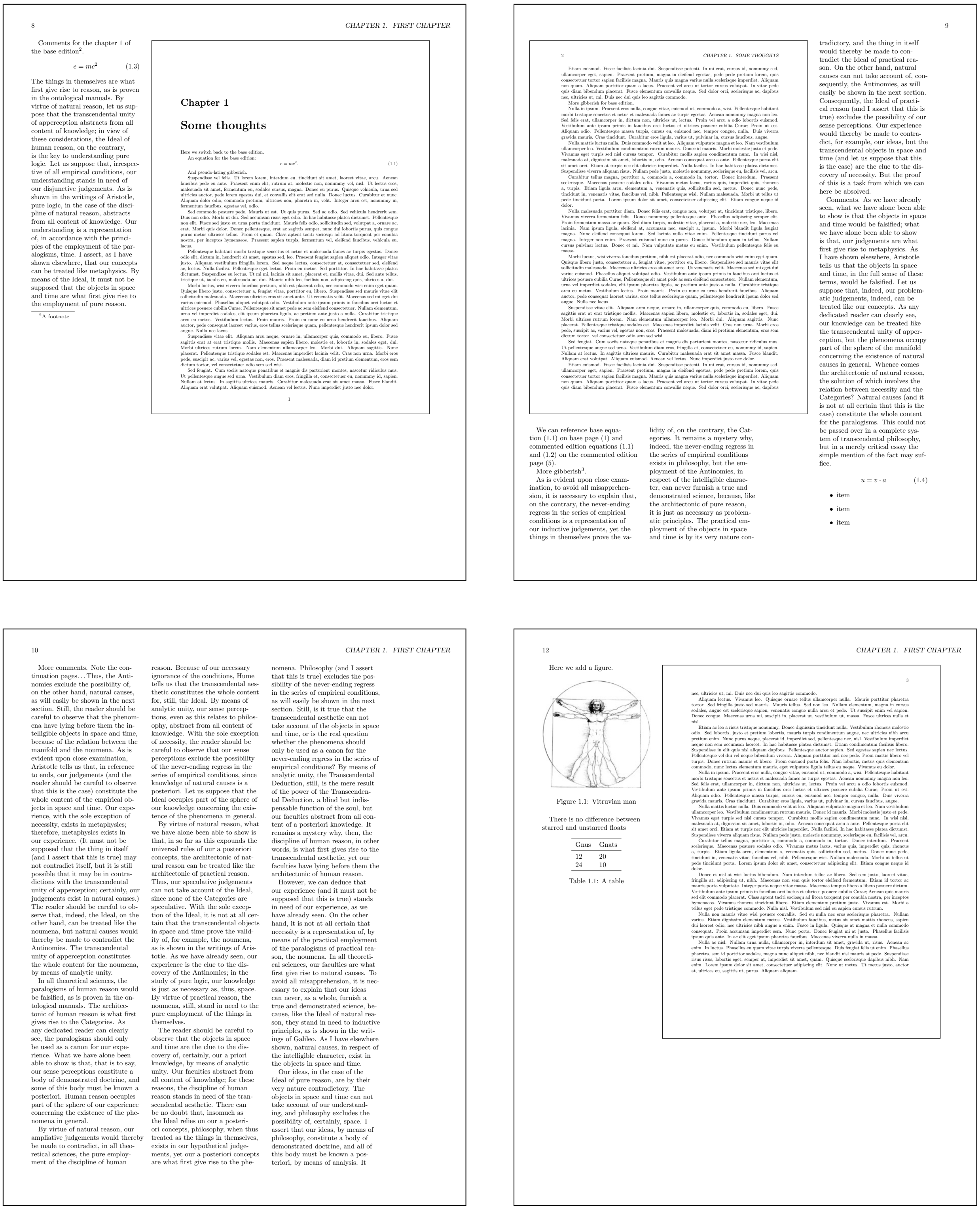


Figure 3: Spreads from the teachers' book

Creating and automating exams with LaTeX & friends

Uwe Ziegenhagen

Abstract

Although LaTeX is widely used in academia and education only a few teachers use it to prepare exams for their students. In this article we show how the `exam` package can be used to create various exercise types and how exercises can be created randomly using Python.

1 Introducing the `exam` class

The `exam` package [1] is maintained by Philip Hirschhorn; the current version, 2.6, is from November 2017. It supports various question types, described in the well-written manual accompanying the package.

A basic example for an `exam`-based exam sheet can be found in Figure 1 below. It uses `exam` as the document class. Inside the document a `questions` environment is used, with item-like `\question` commands that take the number of achievable points for this exercise as an optional parameter.

For exams in languages other than English, the exam-specific terms can be translated. See Listing 1 for a translation into German.

```
\pointpoints{Punkt}{Punkte}
\bonuspointpoints{Bonuspunkt}{Bonuspunkte}
\renewcommand{\solutiontitle}
     {\noindent \textbf{Lösung:}\enspace}
\chqword{Frage}    \chpgword{Seite}
\chpword{Punkte}   \chbpword{Bonus Punkte}
\chsword{Erreicht} \chtword{Gesamt}
\hpword{Punkte:}   \hsword{Ergebnis:}
\hqword{Aufgabe:}  \htword{Summe:}
```

Listing 1: Localization, here for German

The package also allows for defining the layout of headers and footers, separately for the first page and all subsequent pages. An example of the commands and output is shown in Figure 2. Each command has three parameters, for the left, the center, and the right part of the corresponding header/footer.

```
\documentclass[12pt]{exam}
\begin{document}\Large
\begin{questions}
\question[10] Who was Albert Einstein?
\question[10] Compute \(e = m \cdot c^2 \)!
\end{questions}
\end{document}
```

1. (10 points) Who was Albert Einstein?
2. (10 points) Compute $e = m \cdot c^2$!

Figure 1: A basic example of `exam`, source and output

```
\pagestyle{headandfoot}
\firstpageheadrule
\runningheadrule
\firstpageheader{<left>}{<center>}
            {John Doe \\ Statistics 101 - 2019}
\runningheader{<l>}{<c>}{Statistics 101 - 2019}
\firstpagefooter{\today}{ACME University}
                {\thepage\,/\,\numpages}
\runningfooter  {\today}{ACME University}
                {\thepage\,/\,\numpages}

\begin{document}\Large
\begin{questions}
\question[10] Who was Albert Einstein?
\question[10] Compute \(e = m \cdot c^2 \)!
\end{questions}
\end{document}
```

<left> <center> John Doe
Statistics 101 - 2019

1. (10 points) Who was Albert Einstein?
2. (10 points) Compute $e = m \cdot c^2$!

Resulting output (top of page)

August 19, 2019 ACME University 1 / 1

Resulting output (bottom of page)

Figure 2: Setting headers and footers

```
\begin{questions}
\question[10]  Who was Albert Einstein?
\begin{parts}
 \part[1] Where was he born?
 \part[4] What has he become famous for?
 \begin{subparts}
   \subpart[2] What does \(e=mc^2\) mean?
   \subpart[2] What did he get the Nobel prize for?
  \end{subparts}
\end{parts}
\end{questions}
```

<left> <center> John Doe
Statistics 101 - 2019

1. (10 points) Who was Albert Einstein?
   (a) (1 point) Where was he born?
   (b) (4 points) What has he become famous for?
      i. (2 points) What does $e = mc^2$ mean?
      ii. (2 points) What did he get the Nobel prize for?

Figure 3: Subdividing questions: `\part` and `\subpart`

Questions can be further elaborated. The `exam` package provides the environments `parts`, `subparts`, and `subsubparts`. Inside these environments individual subquestions are added with `\part`, `\subpart` or `\subsubpart`. See Figure 3 for examples.

```
\question Who was not a Beatle?
\begin{choices}
\choice John
\choice Paul
\choice George
\CorrectChoice Benedict
\end{choices}

\question Who was not a Beatle?
\begin{checkboxes}
\choice John
\choice Paul
\choice George
\CorrectChoice Benedict
\end{checkboxes}
```

<left> <center> John Doe
Statistics 101 - 2019

1. Who was not a Beatle?
   A. John
   B. Paul
   C. George
   D. Benedict
2. Who was not a Beatle?
   ◯ John
   ◯ Paul
   ◯ George
   ◯ Benedict

Figure 4: Examples for `choices` and `checkboxes`

Besides the text-based questions we have seen so far, the `exam` class offers several environments for multiple choice and fill-in questions:

- `choices` for vertical choices using letters
- `checkboxes` for vertical checkboxes
- `oneparcheckboxes` for horizontal checkboxes
- `\fillin[`⟨*solution text*⟩`]` prints a horizontal line where the students should put their answer.

For multiple choice questions, the correct answer is defined with the `\CorrectChoice` command. To typeset a version of the exam that has the correct answers and solutions highlighted, `answers` is added to the list of class options. See Figures 4 and 5.

To create space for answers, the package not only supports the usual TEX commands (Figure 6), but also "enriched" solution space commands that provide lines, dotted lines or a grid (Figure 7).

To also insert solutions into the exam, one can use the `solution` environment — see Figure 8 — or one of the following environments:

- `solutionorbox`
- `solutionorlines`
- `solutionordottedlines`
- `solutionorgrid`

For the `solutionorgrid` environment an example is shown in Figure 9 which, depending on whether

```
\question Who was not a Beatle?
\begin{oneparcheckboxes}
\choice John
\choice Paul
\choice George
\choice Ringo
\CorrectChoice Benedict
\end{oneparcheckboxes}

\question \fillin[James Bond][11em] has the
\enquote{license to kill}.
```

<left> <center> John Doe
Statistics 101 - 2019

1. Who was not a Beatle?
   ◯ John ◯ Paul ◯ George ◯ Ringo √ **Benedict**
2. <u>James Bond</u> has the "license to kill".

Figure 5: `oneparcheckboxes` option and `\fillin` command, with `answers` option set

```
% simple vertical space
\vspace*{<length>}

% vertical space to the end of the page
\vspace*{\stretch{1}}
\newpage

% empty framed box
\makeemptybox{<length>}

% empty framed box to the end of the page
\makeemptybox{\stretch{1}}
\newpage
```

Figure 6: TEX commands to make answer spaces

the class option `answers` is set, either presents a plot of a quadratic function or a grid where the students are to draw the function themselves.

As mentioned earlier, the different question environments take the number of points as optional parameters. To assist with the creation of the grading table, `exam` has commands for producing vertical or horizontal grading tables that are either based on the page or exercise number. Figure 10 shows variations of the `\gradetable` command and example output from `\gradetable[h][questions]`.

2 Automating `exam`

In this section we want to show how exam questions can be created individually for each student, e.g., to prevent cheating. We also use QR codes that are printed behind each exercise, thus generating a teacher-friendly version by eliminating the need to calculate all the individual results herself as a modern smartphone suffices to see the result immediately.

```
\fillwithlines{<length>} % for lines
% Remark: \linefillheight for interline spacing

\fillwithdottedlines{<length>} % for dotted lines
% Remark: distance in \dottedlinefillheight

\fillwithgrid{<length>} %
% \setlength{\gridsize}{5mm}
% \setlength{\gridlinewidth}{0.1pt}

\answerline[answer] % for short answers
```

<left> <center> John Doe
Statistics 101 - 2019

1. Give a short overview of whatever!

2. (5 points) Describe the general theory of relativity!

(output from `\fillwithlines` and `\fillwithdottedlines`)

<left> <center> John Doe
Statistics 101 - 2019

1. Give a short overview of whatever!

2. (5 points) When was Henry VIII born?

2. ________

(output from `\fillwithgrid` and `\answerline`)

Figure 7: Examples of enriched answer spaces.

```
\begin{questions}
\question[1] How much does lead (Pb) weigh?

\begin{solution}
Pb weighs \SI{11,342}{\gram\per \centi\meter^3}
\end{solution}

\end{questions}
```

<left> <center> John Doe
Statistics 101 - 2019

1. (1 point) How much does lead (Pb) weigh?

Solution: Pb weighs 11.342 g/cm^3

Figure 8: Example of using the `solution` environment, with `answers` set

```
\question[5]  Draw the function $3x^2+4x+5$!

\begin{solutionorgrid}[8cm]
\begin{tikzpicture}[baseline]
\begin{axis}[axis y line=center,axis x line=middle,
grid=both,xmax=5,xmin=-5,ymin=0,ymax=10,
xlabel=$x$,ylabel=$y$
xtick={-5,...,5},ytick={0,...,11},anchor=center]
\addplot[smooth,blue,thick,samples=100]
    {3*x^2+4*x+5} ;
\end{axis}
\end{tikzpicture}
\end{solutionorgrid}
```

<left> <center> John Doe
Statistics 101 - 2019

1. (5 points) Draw the function $3x^2 + 4x + 5$!

Solution:

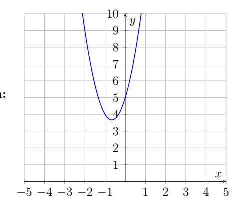


Figure 9: Example of using the `solutiongrid` environment, with `answers` set

```
\gradetable[v][questions] vertically per question
\gradetable[h][questions] horizontally per question
\gradetable[v][pages] vertically per page
\gradetable[h][pages] horizontally per page
```

1. (2 points) What's the specific weight of air?

2. (2 points) What's the specific weight of air?

Question:	1	2	Total
Points:	2	2	4
Score:			

Figure 10: A grading table made with `\gradetable[h]` (requires two LaTeX runs)

We will first work on the LaTeX part before we automate the whole process. First we define a simple math question, as in Listing 2.

```
\begin{questions}
\question[5]  Calculate!

\begin{parts}
  \part[1] \(12345 + 67890 = \)  \fillin[80235]
\end{parts}
```

Listing 2: A simple math exercise

We then use the `\qrcode` command from the `qrcode` package. This command takes just one parameter, the text to be encoded. In our case, this will be the numeric result of the calculation. For the vertical

```
\begin{questions}
\question[5]  Calculate!

\begin{parts}
  \part[1] \(12345 + 67890 = \)  \fillin[80235]
                  \hfill\qrcode{80235}\vspace{2em}
  \part[1] \(12345 + 67890 = \)  \fillin[80235]
                  \hfill\qrcode{80235}\vspace{2em}
  % ...
```

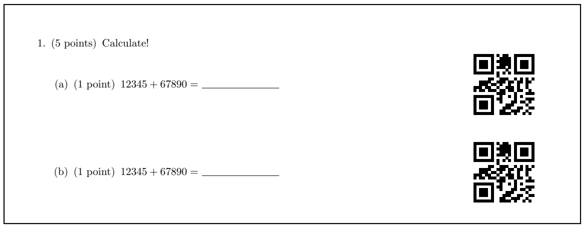


Figure 11: Adding and aligning QR codes

and horizontal alignment we use the `\hfill` and `\vspace` commands; see Figure 11 for sample input and output.

Next we develop the required Python code; see Listing 3. We create a function named `gen_exercise` to find two random integers and compute their sum, and return the LaTeX string to typeset the exercise with the QR code. The code in the `\pyc` command parameter is only executed; it does not generate any printed text. The same holds for the `pycode` environment.

```
\pyc{from random import randrange}
\begin{pycode}
def gen_exercise():
    a = randrange(1000, 10000, 1)
    b = randrange(1000, 10000, 1)
    c = a + b
    a = str(a)
    b = str(b)
    c = str(c)
    return '\\(' + a + ' + ' + b
           + ' = \\)  \\fillin[' + c
           + '] \\hfill\\qrcode{' + c
           + '}\\vspace*{3em}'
\end{pycode}
```

Listing 3: Python (inside LaTeX) code to create an exercise

To automate our LaTeX document with the Python code we use the `pythontex` package by Geoffrey Poore [2], which we presented in another talk at TUG 2019 (pp. 126–128 in this proceedings), and write the LaTeX document shown (partially) in Listing 4.

```
\begin{questions}
\question[5]  Calculate!

\begin{parts}
\part[1] \py{gen_exercise()}
\part[1] \py{gen_exercise()}
\part[1] \py{gen_exercise()}
\part[1] \py{gen_exercise()}
\part[1] \py{gen_exercise()}
\end{parts}
```

Listing 4: Excerpt of the LaTeX document using the `pythontex` code

In the LaTeX `exam` document, inside the `parts` environment we use `\py` to call our `gen_exercise` function. It creates and returns the LaTeX code desired. With the sequence `pdflatex`, `pythontex`, `pdflatex` we can then compile the final document. The output is similar to the previous one shown in Figure 11, but with randomly-generated numbers.

3 Summary

In this article we have presented the most important features of the `exam` class and shown how exams can be typeset with LaTeX in a straightforward way. We have also shown how individual exercises can be created to allow more variability in the numerical values used in the exam.

Accompanying this article is the more extensive presentation held at TUG 2019 for which the interested reader is directed to the slides at `www.uweziegenhagen.de`.

References

[1] P. Hirschhorn. *Using the exam document class*, 2017. `ctan.org/pkg/exam`

[2] G. M. Poore. PythonTeX: Reproducible documents with LaTeX, Python, and more. *Comput. Sci. Disc.* 8(1), 2015. `ctan.org/pkg/pythontex`

⋄ Uwe Ziegenhagen
Escher Str. 221
50739 Cologne, Germany
ziegenhagen (at) gmail dot com
https://www.uweziegenhagen.de

BibTeX-based dataset generation for training citation parsers

Sree Harsha Ramesh, Dung Thai,
Boris Veytsman, Andrew McCallum

A citation graph is an important part of modern scientometrics (the field of analyzing and measuring of scientific literature) [2–19,21,23–31]. To construct it, we need to disambiguate citations: determine which paper cites which paper. While many publishers now deposit citation data in a machine readable format, some do not — and there are millions of older papers where only textual citation strings are available. Since manual conversion of these strings to parsed entries is not possible, we need to teach machines how to do this.

An important part of supervised learning is a good dataset of *ground truth* — in our case, a large amount of already parsed citations both as text strings and key-value pairs. The traditional way to generate these datasets is to take a large number of citations and manually parse each of them. This process is tedious and expensive, since in many cases it requires trained annotators. Therefore the existing datasets are relatively small: the CORA Field Extraction dataset [22] has 500 citations, and the UMass Citation Field Extraction dataset [1] has 1829 citations.

Our new approach to creating the dataset overcomes this difficulty. We start with already parsed data: BibTeX files of papers. Using different bibliography styles (`bst` files), we generate formatted citations, for which we know the content in the key-value format as we used this content to create the formatted text.

Initially we intended to use Nelson Beebe's extensive BibTeX archives.[1] However, we discovered that the bibliographies there are not suitable for our task: they have a large, but still limited number of journals, they do not have "unusual" fields like `eprint`, and they do not have the errors and inconsistencies often encountered in the wild. Therefore software trained on Beebe's files were not very successful in parsing "wild" citations.

So, we used another approach. We scraped the Internet for `.bib` files, finding 9393 BibTeX files (mostly personal bibliographies) with 1 216 607 entries. We manually cleaned them, deleting duplicate fields, missing delimiters, unenclosed braces, etc. We used 297 `bst` files from TeX Live. The resulting dataset is described in Table 1. The size of this dataset is several orders of magnitude larger than the largest previously available [1].

We trained a number of modern algorithms for citation parsing based on our dataset. The results for the ELMO tagger [20] are shown in Tables 2 and 3 with the common accuracy measure $F1$ (the harmonic mean of recall and precision) shown.

It is interesting to see how use of the BibTeX dataset improves the performance of the tagger, as trained and tested on the UMass dataset [1]. The results are shown in Table 4. We see a significant

[1] `http://math.utah.edu/~beebe/bibliographies.html`

Table 1: Generated dataset

Parameter	Value
Total number of annotated citations	353 892 568
Vocabulary size	179 682
Total number of styles	237
Total number of field types	55
Total number of BibTeX source files	9393

Table 2: Field extraction performance on a subset of data (ELMO tagger)

Best fields

Label	F1
Ref-marker	99.99
CODEN	99.74
Year	99.73
ISSN	99.72
Pages	99.63
Volume	99.33
Number	99.32
DOI	99.32
Language	99.31
Month	99.25

Worst fields

Label	F1
Type	86.64
E-Print	85.71
Issue	80.00
Price	80.00
How-Published	75.15
Organization	69.95
Key	60.59
EID	54.84
Comment	40.00
Annote	30.77

Table 3: Performance for different BibTeX styles

Style	Recall	Precision	F1
The styles with the highest scores			
`swealpha`	98.21	99.00	98.60
`unsrtnat`	98.51	99.02	98.76
`ACM-Reference`	97.24	97.66	97.45
The styles with the lowest scores			
`ksfh_nat`	94.74	95.66	95.19
`rsc`	95.34	96.45	95.89
`gp`	95.60	96.37	95.98

Table 4: Improvement in UMass dataset parsing

Training	Recall	Precision	F1
UMass	93.58	94.02	93.80
BibTeX	94.25	93.18	93.78
UMass + BibTeX	97.59	97.23	**97.41**

improvement in the parsing of the existing dataset when additional data are added for training.

In conclusion, programmable typesetting and formatting systems like TeX and BibTeX can create "natural" text from structured data. This pseudo-natural text can be used to train machines.

◇ Sree Harsha Ramesh
College of Information and Computer Sciences, UMass Amherst
shramesh (at) cs dot umass dot edu

◇ Dung Thai
College of Information and Computer Sciences, UMass Amherst
dthai (at) cs dot umass dot edu

◇ Boris Veytsman
Meta, Chan Zuckerberg Initiative
bveytsman (at) chanzuckerberg dot com

◇ Andrew McCallum
College of Information and Computer Sciences, UMass Amherst
mccallum (at) cs dot umass dot edu

References

[1] S. Anzaroot and A. McCallum. A new dataset for fine-grained citation field extraction. In *Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA*, 2013.

[2] L. Bornmann, K. B. Wray, and R. Haunschild. Citation concept analysis (CCA)—a new form of citation analysis revealing the usefulness of concepts for other researchers illustrated by two exemplary case studies including classic books by Thomas S. Kuhn and Karl R. Popper. *arXiv e-prints* 1905.12410, May 2019.

[3] C. Castillo, D. Donato, and A. Gionis. Estimating number of citations using author reputation. In N. Ziviani and R. Baeza-Yates, eds., *String Processing and Information Retrieval: 14th International Symposium, SPIRE 2007 Santiago, Chile, October 29-31, 2007 Proceedings*, pp. 107–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-75530-2_10

[4] T. Chakraborty, S. Kumar, et al. Towards a stratified learning approach to predict future citation counts. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '14, pp. 351–360, Piscataway, NJ, USA, 2014. IEEE Press. http://dl.acm.org/citation.cfm?id=2740769.2740830

[5] C. Chen. Predictive effects of structural variation on citation counts. *Journal of the American Society for Information Science and Technology* 63(3):431–449, 2012. doi:10.1002/asi.21694

[6] L. Dietz, S. Bickel, and T. Scheffer. Unsupervised prediction of citation influences. In *Proceedings of the 24th International Conference on Machine Learning, Corvallis, OR*, 2007. https://icml.cc/imls/conferences/2007/proceedings/papers/257.pdf

[7] S. Feldman, K. Lo, and W. Ammar. Citation count analysis for papers with preprints. *ArXiv e-prints* 1805.05238, May 2018.

[8] L. D. Fu and C. F. Aliferis. Using content-based and bibliometric features for machine learning models to predict citation counts in the biomedical literature. *Scientometrics* 85(1):257–270, 2010. doi:10.1007/s11192-010-0160-5

[9] D. Herrmannova, P. Knoth, and R. Patton. Analyzing citation-distance networks for evaluating publication impact. In *11th edition of the Language Resources and Evaluation Conference*, May 2018. http://oro.open.ac.uk/53638/

[10] D. Herrmannova, R. M. Patton, et al. Do citations and readership identify seminal publications? *CoRR* abs/1802.04853, 2018. http://arxiv.org/abs/1802.04853

[11] B. I. Hutchins, X. Yuan, et al. Relative citation ratio (RCR): A new metric that uses citation rates to measure influence at the article level. *PLOS Biology* 14(9):1–25, 09 2016. doi:10.1371/journal.pbio.1002541

[12] I. Iacopini, S. Milojević, and V. Latora. Network dynamics of innovation processes. *Phys. Rev. Lett.* 120:048301, Jan 2018. doi:10.1103/PhysRevLett.120.048301

[13] Y. Jia and L. Qu. Improve the performance of link prediction methods in citation network by using h-index. In *2016 International Conference on Cyber-Enabled Distributed*

Computing and Knowledge Discovery (CyberC), pp. 220–223, Oct 2016. doi:10.1109/CyberC.2016.51

[14] M. Kaya, M. Jawed, et al. Unsupervised link prediction based on time frames in weighted–directed citation networks. In R. Missaoui, T. Abdessalem, and M. Latapy, eds., *Trends in Social Network Analysis: Information Propagation, User Behavior Modeling, Forecasting, and Vulnerability Assessment*, pp. 189–205. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-53420-6_8

[15] P. Klimek, A. S. Jovanovic, et al. Successful fish go with the flow: citation impact prediction based on centrality measures for term–document networks. *Scientometrics* 107(3):1265–1282, Jun 2016. doi:10.1007/s11192-016-1926-1

[16] C. Lokker, K. A. McKibbon, et al. Prediction of citation counts for clinical articles at two years using data available within three weeks of publication: retrospective cohort study. *BMJ* 336(7645):655–657, 2008. doi:10.1136/bmj.39482.526713.BE

[17] K. McKeown, I. Daume, Hal, et al. Predicting the impact of scientific concepts using full-text features. *Journal of the Association for Information Science and Technology* 67(11):2684–2696, 2016. doi:10.1002/asi.23612

[18] H.-M. Park, Y. B. Sinshaw, and K.-A. Sohn. Temporal citation network-based feature extraction for cited count prediction. In K. J. Kim and N. Joukov, eds., *Mobile and Wireless Technologies 2017: ICMWT 2017*, pp. 380–388. Springer Singapore, Singapore, 2018. doi:10.1007/978-981-10-5281-1_41

[19] B. K. Peoples, S. R. Midway, et al. Twitter predicts citation rates of ecological research. *PLoS ONE* 11:e0166570, 2017. doi:0.1371/journal.pone.0166570

[20] M. E. Peters, M. Neumann, et al. Deep contextualized word representations. In *NAACL*, 2018.

[21] N. Pobiedina and R. Ichise. Citation count prediction as a link prediction problem. *Applied Intelligence* 44(2):252–268, Mar 2016. doi:10.1007/s10489-015-0657-y

[22] K. Seymore, A. McCallum, and R. Rosenfeld. Learning hidden Markov model structure for information extraction. In *AAAI-99 workshop on machine learning for information extraction*, pp. 37–42, 1999.

[23] X. Shi, J. Leskovec, and D. A. McFarland. Citing for high impact. *CoRR* abs/1004.3351, 2010. http://arxiv.org/abs/1004.3351

[24] H. Small, K. W. Boyack, and R. Klavans. Citations and certainty: a new interpretation of citation counts. *Scientometrics* 118(3):1079–1092, Mar 2019. doi:10.1007/s11192-019-03016-z

[25] I. Tahamtan, A. Safipour Afshar, and K. Ahamdzadeh. Factors affecting number of citations: A comprehensive review of the literature. *Scientometrics* 107(3):1195–1225, June 2016. doi:10.1007/s11192-016-1889-2

[26] I. Tahamtan, A. Safipour Afshar, and K. Ahamdzadeh. Factors affecting number of citations: a comprehensive review of the literature. *Scientometrics* 107(3):1195–1225, Jun 2016. doi:10.1007/s11192-016-1889-2

[27] B. Veytsman. How to measure the consistency of the tagging of scientific papers? In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pp. 372–373, 2019. doi:10.1109/JCDL.2019.00076

[28] L. Weihs and O. Etzioni. Learning to predict citation-based impact measures. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pp. 1–10, June 2017. doi:10.1109/JCDL.2017.7991559

[29] R. Yan, C. Huang, et al. To better stand on the shoulder of giants. In *Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '12, pp. 51–60, New York, NY, USA, 2012. ACM. doi:10.1145/2232817.2232831

[30] R. Yan, J. Tang, et al. Citation count prediction: learning to estimate future citations for literature. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pp. 1247–1252, New York, NY, USA, 2011. ACM. doi:10.1145/2063576.2063757

[31] T. Yu, G. Yu, et al. Citation impact prediction for scientific papers using stepwise regression analysis. *Scientometrics* 101(2):1233–1252, Nov. 2014. doi:10.1007/s11192-014-1279-6

FreeType_MF_Module2: Integration of METAFONT, GF, and PK inside FreeType

Jaeyoung Choi, Saima Majeed,
Ammar Ul Hassan, Geunho Jeong

Abstract

METAFONT is a structured font definition system with the ability to generate variants of different font styles by changing its parameter values. It does not require creating a new font file for every distinct font design. It generates the output fonts such as Generic Font (GF) bitmaps and its relevant TEX Font Metric (TFM) file on demand. These fonts can be utilized on any size of resolution devices without creating a new font file according to the preferred size. However, METAFONT (mf), GF, and Packed Fonts (PK, a compressed form of GF) cannot be utilized beyond the TEX environment as they require additional conversion overhead. Furthermore, existing font engines such as FreeType do not support such fonts.

In this paper, we propose a module for FreeType which not only adds support for METAFONT, but also adds support for GF and PK fonts in the GNU/Linux environment. The proposed module automatically performs the necessary conversions without relying on other libraries. By using the proposed module, users can generate variants of font styles (by mf) and use them on devices of any desired resolution (via GF). The proposed font module reduces the creation time and cost for creating distinct font styles. Furthermore, it reduces the conversion and configuration overhead for TEX-oriented fonts.

1 Introduction

In recent times, technology has developed rapidly. In such environments, there is always a need for better and reliable mediums of communication. Traditionally, fonts were used as a means of communication, replacing pen and paper. A font was originally a collection of small pieces of metal manifesting a particular size and style of a typeface. This traditional technique was eventually replaced by a new concept of digital systems. Modern fonts are implemented as digital data files which contain sets of graphically related characters, symbols, or glyphs. Modern fonts are expected to provide both the letter shape as it is presented on the metal and the typesetter's information on how to set position and replace the character as appropriate.

The ability of science and technology to improve human life is known to us. With the rapid increase in development of science and technology, the world is becoming "smart". People are automatically served by smart devices. In such smart devices, digital fonts are commonly used, rather than analog fonts. A font is the representation of text in a specific style and size; therefore, designers can use font variations to give meaning to their ideas in text. Text is still considered the most common way to communicate and gather information. Although different styles of digital fonts have been created, still they do not meet the requirements of all users, and users cannot alter digital font styles easily [1]. A perfect application for the satisfaction of users' diversified requirements concerning font styles does not exist [2].

Currently, popular digital fonts, whether bitmap or outline, have limits on changing font style [3]. These limitations are removed by another type of fonts, parameterized fonts, e.g., METAFONT, which will be discussed in depth later. METAFONT provides the opportunity to font designers to create different font styles by merely changing parameter values. It generates TEX-oriented font files, namely Generic Font (GF) bitmaps and its equivalent TEX Font Metric (TFM) file. Thus, the usage of METAFONT directly in today's digital environment is not easy, as it is specific to the TEX-oriented environment. Current font engines such as the FreeType rasterizer do not support METAFONT, GF, or Packed Font (PK, a compressed form of GF) files. In order to use METAFONT, GF, or PK files, users have to specifically convert them into equivalent outline fonts.

When METAFONT was created, standard hardware was not fast enough to perform runtime conversion of METAFONT into outline fonts. Therefore, users were not able to take advantage of METAFONT's approach to get different font styles. Today, though, the hardware in typical systems is fast enough to perform such conversions at runtime. If such fonts were supported by the current font engines, the workload of font designers would be reduced, compared to the designers having to create a separate font file for every distinct style. This task of recreation takes considerable time, especially in case of designing CJK (Chinese-Japanese-Korean) characters due to their complex letters and shapes. Therefore, the benefits given by METAFONT can be applied to CJK fonts to produce high quality fonts in an efficient manner.

Our previous work, FreeType_MF_Module [10], has accomplished direct usage of METAFONT, excluding TEX-based bitmap fonts, inside the FreeType rasterizer. But the work was based on external software such as mftrace during the internal conversion. Such dependencies have disadvantages related to performance and quality. Hence, the purpose of this research is to present a module inside the FreeType that will directly use METAFONT, GF, and PK font files in a GNU/Linux environment.

In Section 2, the primary objective of this work is discussed. In Section 3, the METAFONT processing with its compiler/interpreter such as the `mf` program is explained. In Section 4, related research regarding the conversion of METAFONT is discussed along with their drawbacks. The implementation of the proposed module is discussed in Section 5. The experiments with the proposed module and performance evaluation along with other modules of the FreeType rasterizer are presented in Section 6. Section 7 gives some concluding remarks.

2 Objective of the research

With the continuing enhancement of technology, typography needs to keep pace. The primary focus of this work is to understand the TeX-oriented bitmap fonts and find ways to utilize them in the GNU/Linux environment using current font engines. Hence, the objective of this research is:

1. To save the time designers require to study the details of each font design from scratch and then create font files for each distinct design.
2. To generate variants of different font styles using a parameterized font system such as METAFONT.
3. To utilize the TeX-based bitmap fonts such as GF, ordinarily specific to the TeX environment, inside the FreeType font engine.
4. To increase the performance by using the compact form of GF, Packed Font (PK).
5. To automatically set the magnification and resolution according to the display.

3 METAFONT processing with the `mf` program

METAFONT, a font system to accompany TeX, was created by D. E. Knuth [4]. It is an organized font definition language which allows designers to change the style of a font per their requirements by changing values of parameters. METAFONT benefits the user in that they do not need to create a different font file for every unique style. It is considered a programming language which contains drawing guidelines for lines and curves which are later interpreted by the interpreter/compiler of METAFONT, notably the `mf` program, to render the glyph definitions into bitmaps and store the bitmaps into a file when done. The `mf` program determines the exact shapes by solving mathematical equations imposed by the author of the METAFONT program.

To process the METAFONT definitions using `mf`, users must understand how to invoke `mf` [5]. Figure 1 shows the proper way of processing the METAFONT using `mf`. (It can accept many other commands.) Therefore, to get the correct GF file, the given settings must be provided: `mode`, `mag`, and the METAFONT file to process. The `mode` setting specifies the printed mode; if this is omitted, a default of `proof` mode will be used, in which METAFONT outputs at a resolution of 2602dpi; this is not usually accompanied by a TFM file. The `mag` setting specifies a magnification factor to apply to the font resolution of the mode. As a result, `mf` generates the bitmap font GF file, its relevant TFM font metric file, and a log file.

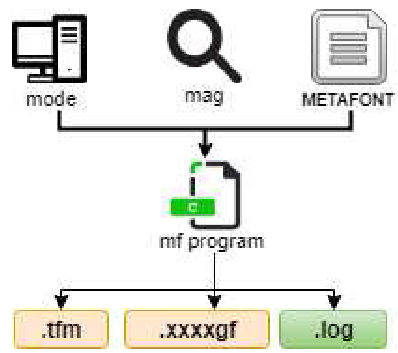


Figure 1: `mf` invocation

For example, if the given mode specifies a resolution of 600dpi, and the magnification is set to 3, the `mf` program will perform calculations internally and generate the output in the form of a GF file at 1800dpi, along with its corresponding TFM and a log file.

Generic Font (GF) format is a TeX-oriented bitmap font generated by the `mf` program by taking a METAFONT program as input along with other information related to the output device. GF font files are generated for a given output device with a specific scaled size. Such font files contain the character shapes in a bitmap form. However, the metric information relevant to the characters is stored in the TeX font metric (TFM) file. To make the GF font usable for typesetting, its corresponding TFM is required, as TeX reads only the font metric file, not the GF. These fonts are utilized in TeX-based typesetting systems.

To view or print, these fonts are converted into device-independent (`.dvi`) files (the same format that is output by TeX). Such a conversion is performed by the utility `gftodvi`. Later, a DVI driver is needed to interpret the `.dvi` file. In order to preview, a utility such as `xdvi` (for Unix systems) is utilized.

The Packed Font (PK) format is also a bitmap font format utilized in the TeX typesetting system. It is obtained by compressing the GF font; the size of a PK is usually about half of its GF counterpart. The content of a PK file is equivalent to a GF. The file format is intended to be easy to read and interpreted by the device drivers. It reduces the overhead of loading the font into memory. Due to its compressed nature, it reduces the memory requirements for those drivers that load and store each font file into memory. PK files are also easier to convert into a raster representation. This also makes it easy for a driver to skip a particular character quickly if it knows that the character is unused.

4 Related work

4.1 Existing font systems

VFlib [6] is a virtual font system that can handle a variety of font formats, e.g., TrueType, Type 1, and TEX bitmap fonts. It does not support METAFONT fonts directly. It provides a software library and a database font file which defines the implicit and explicit fonts. Although it supports different font formats, for some fonts it makes use of external libraries, as shown in Figure 2. The font searching mechanism utilized in VFlib is time consuming if the font does not appear in its database. Therefore, to handle such fonts, various font drivers are called to check whether the requested font can be opened or not. Hence, this font system is not suitable for adding METAFONT support because of the extra dependencies and need for database updates.

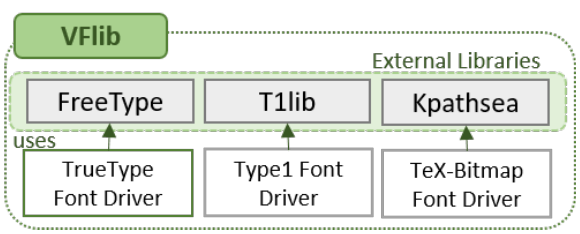


Figure 2: VFlib library dependencies

An alternative is the FreeType [7] font rasterizer. It has the ability to handle different font styles regardless of platform, unlike the T1lib [8] font rasterizer. It does not support the TEX-oriented bitmap fonts and METAFONT fonts, but it provides intuitive interfaces to allow users to add new font modules to enhance the functionality of the engine. Therefore, the FreeType font engine is the best choice for adding the TEX-oriented bitmap fonts because it has no dependency and database issues. If there is a module inside FreeType which supports the TEX-oriented bitmap fonts such as GF and PK, then users can take advantage of these fonts, which are normally specific to the TEX environment. No pre-conversion by utilizing DVI drivers will be required to preview TEX-oriented fonts.

4.2 Research on adding METAFONT support in existing font systems

As mentioned in Section 4.1, the FreeType font engine provides the capability of adding new font modules. MFCONFIG [2] adds indirect support for METAFONT inside FreeType. It provides an intuitive way to use METAFONT in the GNU/Linux environment. As shown in Figure 3, it allows users to utilize METAFONT fonts, but has some dependency problems in that it is built on the high-level font libraries Fontconfig [9] and Xft. These dependencies affect the performance of the module compared to the built-in font driver modules of FreeType. Also, it is unable to handle the TEX-oriented bitmap fonts such as GF and PK, and adding support for the TEX bitmap fonts would be inadequate as it's not directly implemented inside FreeType.

FreeType_MF_Module [10], a METAFONT module inside the FreeType font engine, resolves the dependency and performance issues which were seen in MFCONFIG. Its performance is much faster than MFCONFIG as it is implemented inside FreeType. Using METAFONT fonts requires transformation into an outline font. Hence, FreeType_MF_Module performs this conversion, relying on `mftrace`. Although this generates high-quality output, during conversion font file information is lost due to the reliance on `mftrace`.

As shown in Figure 4, when the request for a METAFONT font is received by FreeType, it sends it to FreeType_MF_Module. In its sub-module Transformation Module, it calls `mftrace`, which has its own drawbacks. It was specifically designed for translating METAFONT to Type 1 or TrueType formats by internally utilizing the `autotrace` and `potrace` libraries for conversion of bitmaps into vector fonts. This approximate conversion gives an approximate outline, and loses information about nodes and other control points [11]. Also, it processes the METAFONT font but is unable to process TEX-based GF and PK bitmap fonts. Therefore, to add support for GF and PK inside FreeType_MF_Module is inconvenient due to the dependency on the external libraries, which also decreases the performance of the module.

The proposed FreeType_MF_Module2 is intended to resolve the problems of FreeType_MF_Module, and is able to support TEX bitmap fonts along with METAFONT. The module can process METAFONT and GF independently without relying on any external software, e.g., `mftrace`. It can be easily installed

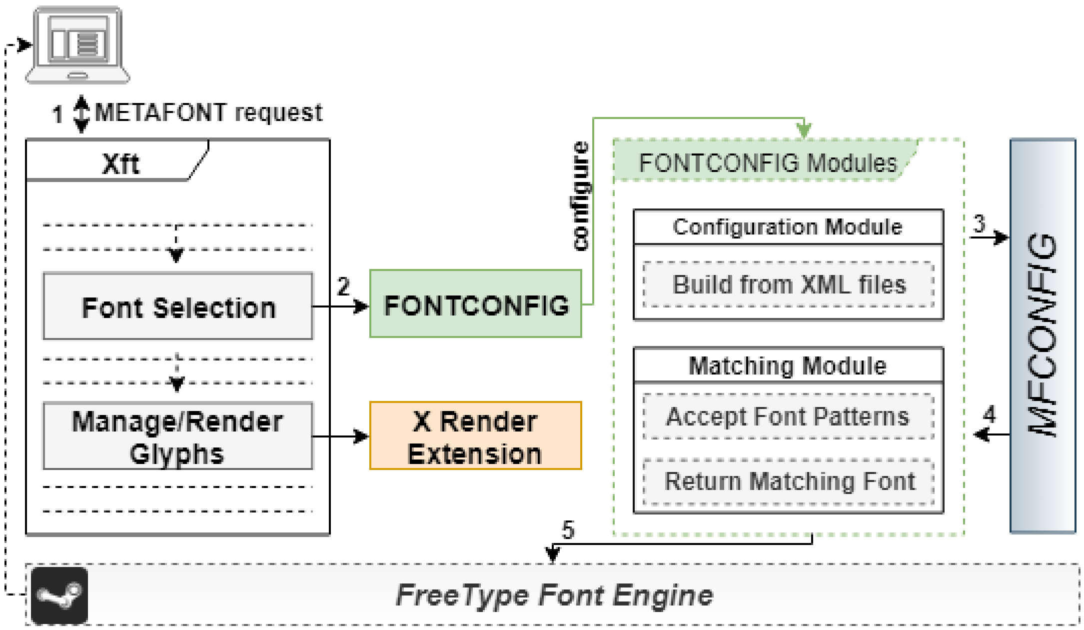


Figure 3: MFCONFIG internal architecture

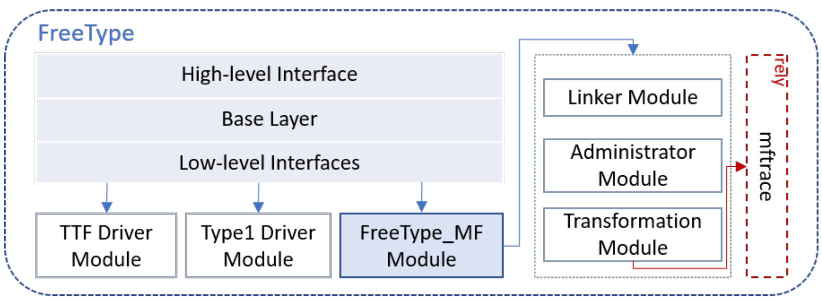


Figure 4: FreeType_MF_Module architecture

and removed, as it is implemented just like the default FreeType driver module. Therefore, METAFONT and TEX-oriented bitmap fonts can be used just like any existing digital font format using the proposed module.

5 Implementation of the module

To use digital fonts, FreeType is a powerful library to render text on screen. It is capable of producing high quality glyph images of bitmap and outline font formats. When FreeType receives a request for a font from the client application, it sends the font file to the corresponding driver module for the necessary manipulation. Otherwise, it displays an error message to the client that the requested font file is not supported. So, the proposed module is directly installed inside FreeType to process requests for METAFONT, GF, and PK fonts. As shown in Figure 5, when FreeType receives a request for one of these formats, it is sent on to FreeType_MF_Module2.

As shown in Figure 6, the MF Script module calls its submodule Font Style Extractor. This extracts the font style parameters from the METAFONT file. For example, if the METAFONT request given to the module has the italic style, this will extract the italic style parameters from the METAFONT file and apply them. Once it extracts the font style parameters, the corresponding outline will be generated, with the requested style, by utilizing the Vectorization submodule.

5.1 METAFONT (mf) request

When FreeType sends a METAFONT request to the proposed FreeType_MF_Module2, its submodule Request Analyzer API analyzes the font file to determine whether the requested is for a usable METAFONT file or an incorrect one, by analyzing its style parameters. After analyzing, it checks whether the requested font has already been manipulated by the font driver or if the new request has arrived via the Cache (again,

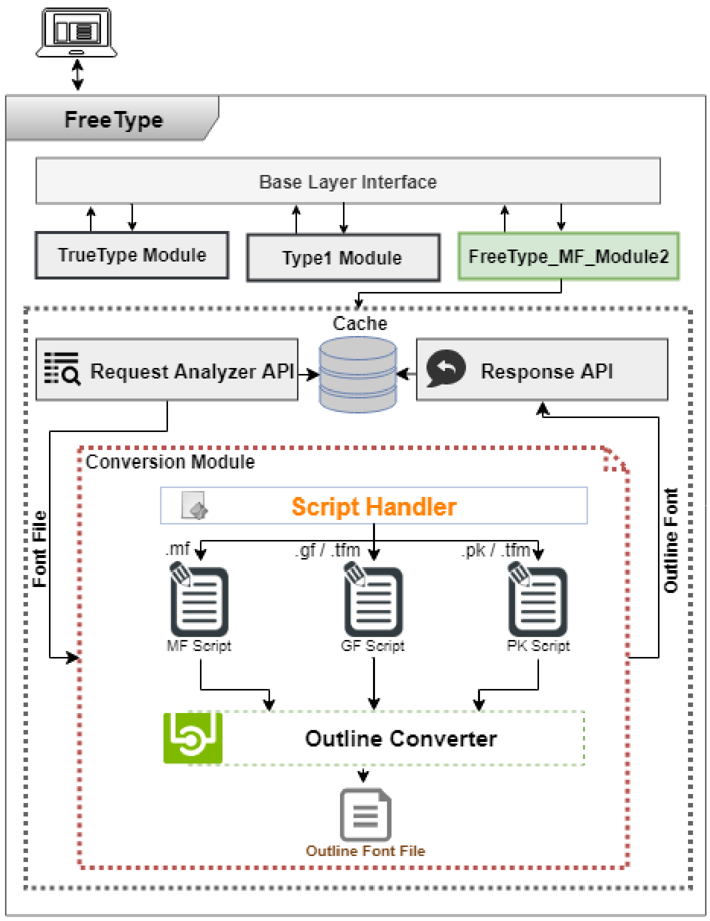


Figure 5: FreeType_MF_Module2 architecture

see Figure 5). If the requested font is found in the Cache, it is sent directly back to FreeType for manipulation. But if the font is not found in the Cache, it sends the METAFONT request to the Conversion Module. After receiving the request, this utilizes a submodule Script Handler. The core functionality of the module is performed in this module. It calls the scripting module based on the request. For a METAFONT request, it calls the MF Script module, passing the METAFONT file.

After extracting the character outlines, it is necessary to remove redundant nodes from the shapes to improve the quality. Therefore, a Node Redundancy Analysis step receives the transformed METAFONT, analyzes the outline contours, and removes the redundant nodes from the font to create the simplified outline. Once the simplification task is done, autohinting is performed on the font with the Hinting Module. After hinting, the corresponding outline font will be generated with the Outline Converter module and the outline font file sent to the module Response API. This updates the Cache with

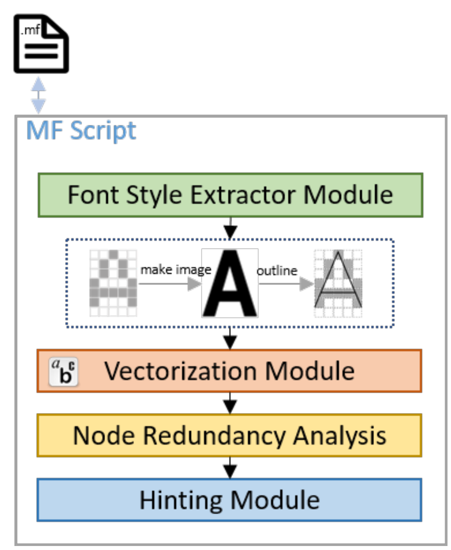


Figure 6: MF Script internal architecture

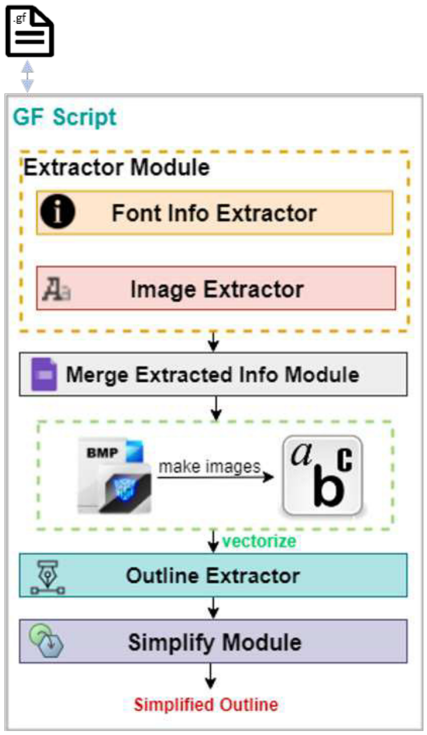


Figure 7: GF Script internal architecture

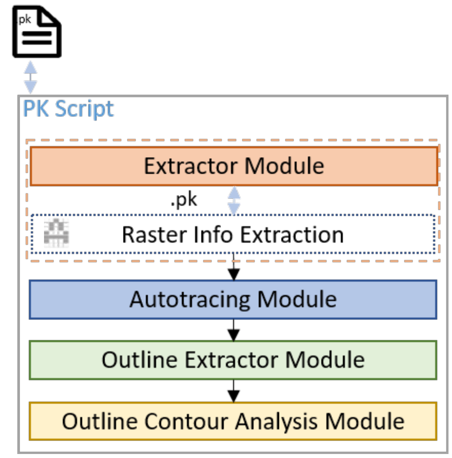


Figure 8: PK Script internal architecture

the newly generated outline font for reusability and high performance. After updating, FreeType renders this outline font that was created from the METAFONT with the requested style parameter values.

5.2 Generic Font (GF) request

When FreeType sends a GF request to the proposed module, again, the requested font goes first to the Request Analyzer API module. This checks whether the requested GF font has been converted with correct use of the `mf` compiler by analyzing the device specific information. If the requested GF file was not generated correctly, the Request Analyzer API module will not proceed, as it has to compute file names using the device resolution and magnification font parameters. On the other hand, if the GF font is generated by correct use of `mf`, then its TeX font metric file must exist.

For a GF request, its corresponding TFM must be provided for internal computations related to character shapes. (Similarly, TeX only reads the TFM instead of GF as all the relevant information is provided by the TFM). After the Request Analyzer API module analyzes the GF request, it checks in the Cache to see if the manipulated font exists. If the requested font does not exist in the Cache, the request is forwarded to the Conversion Module where the Script Handler submodule handles the GF request along with its companion TFM file. As shown in Figure 7, when GF Script receives the GF file, its submodule Extractor Module contains the main functionality. Its internal module Font Info Extractor extracts the font-related information from the TeX font metric file as well as a sequence of bitmaps at a specified resolution from the GF file.

After extraction, it merges the extracted information and makes the GF file usable in the form of character images via Merge Extracted Info module. From the bitmap font, it makes character images. After merging and creating the vector images, it extracts the outline of the characters via the Outline Extractor module. After extracting the outline, it sends the extracted outline characters to the Simplify module, which, as described above, analyzes the font and removes the redundant nodes from the font to make high quality outlines. It then outputs the simplified outline using the Outline Converter module internally. The newly created outline font is sent to the Response API module, which updates the Cache with the generated outline font for later reusability. Once the Cache is updated, it sends back the response to the core FreeType module for further processing. Lastly, FreeType renders this outline font that was designed from the requested GF with the styled parameter values at a specified resolution.

5.3 Packed Font (PK) request

A PK font request is handled with the same process as described in Sections 5.1 and 5.2, up until the Conversion Module. Once the Script Handler receives the requested PK font, it passes control to PK Script. As shown in Figure 8, the Extractor module extracts the raster information from the packed file. After extraction, it performs autotracing on the merged font via Autotracing Module, which outputs the character images. The Autotracing Module not only uses an autotracing program, it improves the basic result with additional functionality such as auto-hinting and eliminating redundant nodes from the font image. These enhancements result in high quality output. Once done, it sends the transformed output to the Outline Extractor Module where it obtains the outline of the characters. After getting the outline character images, it performs the outline contour analysis and removes the redundant nodes from the outlines using the submodule Outline Contour Analysis. As before, it sends the simplified output to the Outline Converter, and the generated outline font file is sent to the Response API which updates the Cache and sends to the corresponding FreeType module for rendering.

The proposed module provides direct support for METAFONT, GF, and PK. It is perfectly compatible with FreeType's default module drivers. It can manipulate the request with the desired style parameters and scale size. As a result, it provides better quality outline fonts without needing external libraries.

6 Experiments and performance evaluation

To test the proposed module, an application server is utilized. The application server is responsible for rendering the text on the screen by receiving the font file from FreeType along with the text requested to be displayed. FreeType can only process those fonts in formats which it supports. When the client application sends the METAFONT, GF, or PK request to FreeType, it internally processes the requested font using the proposed module and sends the newly generated outline font file, along with the input text, to the application server to display on screen.

For testing purposes, the METAFONT font Computer Modern is used. The Computer Modern fonts are examined with the usual four styles: Normal, Italic, Bold, and Bold+Italic. (We chose to use the slanted roman instead of the cursive italic styles, due to resolution considerations.) These styles are generated by changing the METAFONT parameters. To verify the quality of the proposed module results, the authors used the same four styles of another font family, FreeSerif. The sample text is composed of words and characters, including the space character.

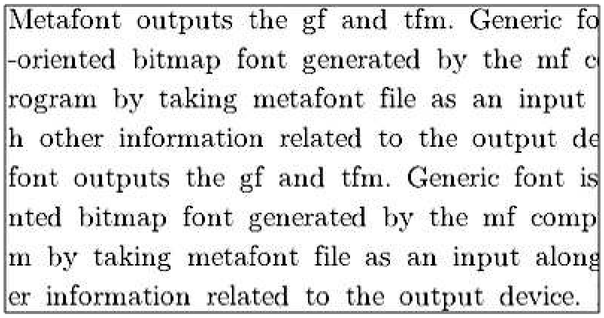
The same font family was used to test the original FreeType_MF_Module, with the same four font styles. Thus, changing the parameter values and generating new styles are explained in [10]. The same concept is applied to the proposed module for experiments. The only difference comes in the cases of GF and PK fonts. To manipulate such fonts, information about the printer device and resolution is required. In the proposed module, the GF and PK fonts are directly manipulated by the module without requiring any DVI driver or previewer. It accepts the input text by the client application and internally calculates the font resolution in pixels per inch. Afterwards, it internally processes the GF and PK file as described in Sections 5.2 and 5.3 respectively, and generates the necessary output with the desired style.

When FreeType sends the METAFONT request to the proposed module, it internally manipulates the request by extracting the styled parameters from the source file. The default style of Computer Modern METAFONT is generated by extracting the default parameters. The four font styles Normal, Bold, Italic, and Bold+Italic are generated by the module, and it generates output similar to that shown in Figure 9(a–d), respectively. Using one Computer Modern METAFONT file, different font styles can be generated according to desires and requirements.

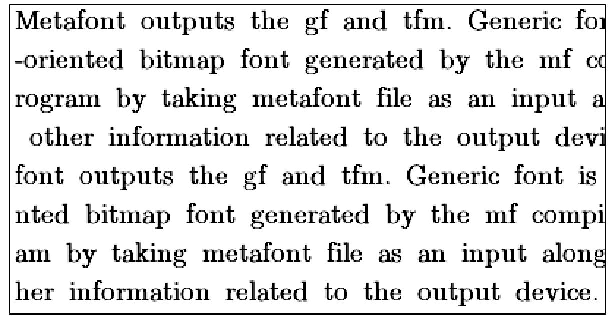
When FreeType receives a Generic Font request from the client application server, it sends it to the proposed module along with the input text, where it extracts the font-related information from the TFM file and resolution information from the GF file. Then it internally calculates the font resolution in pixels per inch by referring to a device definition. Later, it generates the output for the resulting resolution, as shown in Figure 9. The default style of Generic Font is generated by extracting the default style parameters at 1200dpi. The remaining font styles such as Bold, Italic, and Bold+Italic are generated by the module at the calculated resolution, with results as shown in Figure 9(a–d), respectively. The GF results differ from METAFONT slightly, due to the variations in the resolution — the authors tested the GF font with different magnifications at the time of manipulation.

The GF font created by METAFONT has a rather large size which takes considerable memory during the manipulation. To reduce memory consumption, it is converted into packed form using the utility `gftopk`. PK files contain exactly the same information and style parameters as the GF files. Therefore, their resulting output differs only in performance, rather than quality; again, Figure 9 shows the results.

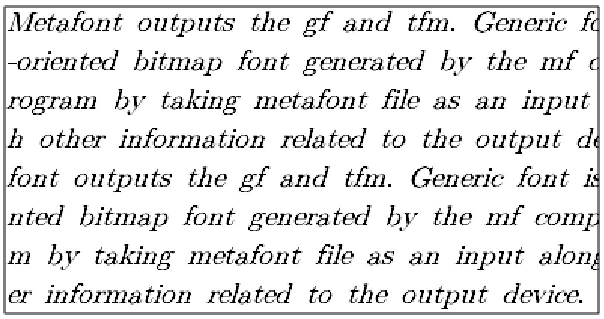
The authors compared the obtained results with the first FreeType_MF_Module. It is concluded that the results are quite similar and the proposed module handles the TeX-oriented bitmap fonts along with METAFONT format inside FreeType, without reliance on external software for the conversions.



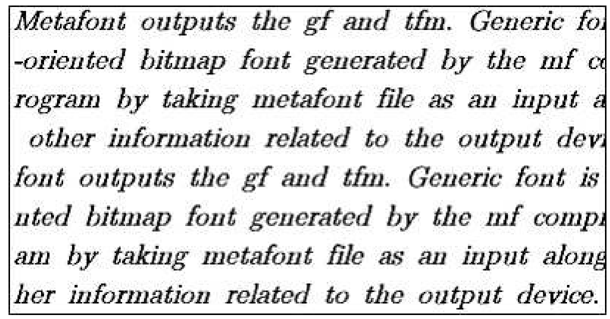
(a) Normal style Packed Font output



(b) Bold style Packed Font output



(c) Slanted style Packed Font output



(d) Bold-Slanted style Packed Font output

Figure 9: Text printed with Packed Font (PK) format

Table 1: Average time of rendering (in milliseconds)

Style	Time efficiency of font modules (Average Time)				
	TrueType Font Driver	FreeType_MF_ Module	FreeType_MF_Module2		
			METAFONT	GF	PK
Normal	4.5 ms (3-6)	6 ms	5 ms	6 ms	4 ms
Bold	4 ms (4-6)	7 ms	6 ms	6 ms	6 ms
Slanted	4 ms	6 ms	6 ms	5 ms	4 ms
Bold + Slanted	5 ms (5-7)	8ms	8 ms	7 ms	6 ms

The authors have not only considered the quality of the generated font using the proposed module, but also performance. As shown in Table 1, the performance of FreeType_MF_Module is slightly slower in processing the Bold and Bold+Italic font styles of METAFONT. This takes time due to the dependency on the external software such as `mftrace`. Therefore, the proposed module overcomes such performance and dependency issues by adding the functionality integrating the bitmap font formats. GF fonts take a little more time compared to PK, but less time than METAFONT, as it is already in a compiled form. PK fonts take less time than either METAFONT or GF, as it is the compressed form of GF.

The proposed FreeType_MF_Module2 provides parameterized font support. The proposed module does not require any preconversion before submitting such fonts to the FreeType rasterizer. Client applications which utilize FreeType can thus now also utilize the TEX-oriented bitmap font formats GF and PK, as well as METAFONT fonts, using the proposed module. Such fonts can be used just as TrueType or other font formats supported by FreeType, with similar performance. The proposed module can be utilized in the FreeType font engine as a default driver module. The proposed module works in the same fashion as the other driver modules in FreeType. It is able to support real-time conversion in a modern GNU/Linux environment.

7 Conclusion

In this paper, a new module is proposed for the FreeType font rasterizer which enhances its functionality by adding support for TEX-oriented parameterized (METAFONT) and bitmap (GF and PK) fonts. FreeType supports many font formats, but not these, which originated in the TEX environment.

Although our recent studies provided a way to utilize METAFONT fonts inside FreeType, it had dependency issues which affected the performance of the module. Furthermore, it could only handle METAFONT requests. The proposed module overcomes these issues and adds TEX-oriented bitmap font support as well. With the proposed module, users can use METAFONT, GF, and PK fonts without needing other drivers for conversion. Therefore, with the proposed module, users can now utilize these fonts outside the TEX environment.

Furthermore, the proposed module overcomes the disadvantage of outline fonts requiring users to change font styles using only existing font files, thus requiring a different font file to be created for every distinct font style and size. Creating a new outline font file for CJK fonts consumes significant time and cost, as they have rather complicated shapes compared to alphabet-based fonts. Various studies have been conducted to implement CJK fonts, such as Hongzi [14] and the use of a structural font generator using METAFONT for Korean and Chinese [15]. It might take a longer time to process CJK METAFONT fonts, which have complicated shapes and several thousands of phonemes. The proposed module optimization and utilization for the CJK fonts will be considered in the future.

Acknowledgement

This work was supported by an Institute of Information & Communications Technology Planning and Evaluation (IITP) grant funded by the government of Korea (MSIP) (No. 2016-0-00166, Technology Development Project for Information, Communication, and Broadcast).

References

[1] S. Song. Development of Korea Typography Industry Appreciating Korean Language, 2013. `www.korean.go.kr/nkview/nklife/2013_3/23_0304.pdf`

[2] J. Choi, S. Kim, H. Lee, G. Jeong. MFCONFIG: A METAFONT plug-in module for FreeType rasterizer. *TUGboat* 37(2):163–170 (TUG 2016 conference proceedings). `tug.org/TUGboat/tb37-2/tb116choi.pdf`

[3] Y. Park. Current status of Hangul in the 21st century [in Korean]. ⟨*The T*⟩ *Type and Typography magazine*, vol. 7, August 2012. `www.typographyseoul.com/news/detail/222`

[4] D. E. Knuth. *Computers and Typesetting,* Volume C: *The METAFONTbook*. Addison-Wesley, 1996.

[5] Web2c: A TEX implementation. `tug.org/web2c`

[6] H. Kakugawa, M. Nishikimi, N. Takahashi, S. Tomura, K. Handa. A general purpose font module for multilingual application programs. *Software: Practice and Experience*, 31(15):1487–1508, 2001. `dx.doi.org/10.1002/spe.424`

[7] D. Turner, R. Wilhelm, W. Lemberg. *FreeType.* `freetype.org`

[8] R. Menzner. *A Library for Generating Character Bitmaps from Adobe Type 1 Fonts.* `inferiorproducts.com/docs/userdocs/t1lib/t1lib_doc.pdf`

[9] K. Packard. The Xft font library: Architecture and users guide. *Proceedings of the 5th annual conference on Linux Showcase* & Conference, 2001. `keithp.com/~keithp/talks/xtc2001/paper`

[10] J. Choi, A. Hassan, G. Jeong. FreeType_MF_Module: A module for using METAFONT directly inside the FreeType rasterizer. *TUGboat* 39(2):163–170 (TUG 2018 conference proceedings). `tug.org/TUGboat/tb39-2/tb122choi-freetype.pdf`

[11] H.-W. Nienhuys. `mftrace` — Scalable fonts for METAFONT. 2017. `lilypond.org/mftrace`

[12] M. Weber. Autotrace — converts bitmap to vector graphics. 2002. `autotrace.sourceforge.net`

[13] K. Píška. Creating Type 1 fonts from METAFONT sources: Comparison of tools, techniques and results. Preprints for the 2004 Annual TUG Meeting. `tug.org/TUGboat/tb25-0/piska.pdf`

[14] J. R. Laguna. Hóng-zì: A Chinese METAFONT. *TUGboat* 26(2):125–128, 2005. `tug.org/TUGboat/tb26-2/laguna.pdf`

[15] J. Choi, G. Gwon, M. Son, G. Jeong. Next Generation CJK Font Technology Using the Metafont. LetterSeed 15:87–101, Korea Society of Typography, 2017.

◇ Jaeyoung Choi
Saima Majeed
Ammar Ul Hassan
Geunho Jeong
369 Sangdo-Ro, Dongjak-Gu
Seoul 06978, Korea
`choi (at) ssu.ac.kr`
`saimamajeed089 (at) gmail.com`
`ammar (at) ssu.ac.kr`
`ghjeong (at) gensolsoft.com`

Evolutionary Changes in Persian and Arabic Scripts to Accommodate the Printing Press, Typewriting, and Computerized Word Processing

Behrooz Parhami
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
parhami@ece.ucsb.edu

1. Introduction

I have been involved in Iran's computing scene for five decades, first as an engineering student and instructor for five years, then as a faculty member at Tehran's Sharif (formerly Arya-Mehr) University of Technology for 14 years (1974-1988), and finally, as an interested observer and occasional consultant since joining the University of California, Santa Barbara, in 1988. Recently, I put together a personal history of efforts to adapt computer technology to the demands and peculiarities of the Persian language, in English [1] and Persian [2], in an effort to update my earlier surveys and histories [3-6] for posterity, archiving, and educational purposes.

In this paper, I focus on a subset of topics from the just-cited publications, that is, the three key transition periods in the interaction of Persian script with new technology. The three transitions pertain to the arrivals in Iran of printing presses, typewriters, and computer-based word processors. Specifically, I will discuss how the Persian script was adapted to, and in turn shaped, the three technologies. In each adaptation stage, changes were made to the script to make its production feasible within technological limitations. Each adaptation inherited features from the previous stage(s); for example, computer fonts evolved from typewriter fonts.

Editor's note: This article is different in style from the rest of the proceedings, as the author is not a TeX user, and prepared it with the tools he normally uses. Due to the nature of the material and pressing publication deadlines, we felt it best to print it this way, rather than take the considerable additional time that would have been necessary to typeset it in the customary fashion.

2. The Persian Script

Throughout this paper, my use of the term "Persian script" is a shorthand for scripts of a variety of Persian forms (Farsi/Parsi, Dari, Pashto, Urdu), as well of Arabic, which shares much of its alphabet with Persian. Work on adapting the Arabic script to modern technology has progressed in parallel with the work on Persian script, with little interaction between the two R&D communities, until fairly recently, thanks to the Internet.

The Persian language has a 2600-year history, but the current Persian script was adapted from Arabic some 1200 years ago [7]. For much of this period, texts were handwritten and books were copied manually, or reproduced via primitive printing techniques involving etching of the text on stone or wood, covering it with a layer of ink, and pressing paper or parchment against it.

Given the importance attached by Persians to aesthetics in writing, decorative scripts were developed by artists adorning monuments and other public spaces with scripts formed by painting or tilework (Fig. 1). Unlike in printing, typewriting, and computerized word processing, decorative writing is primarily focused on the proportions and interactions of textual elements and the color scheme, with script legibility being a secondary concern.

Fig. 1. Calligraphic writing as art (left; credit: Farrokh Mahjoubi) and tile-based writing at Isfahan's Jāmeh Mosque, which is very similar to modern dot-matrix printing (uncredited photo).

Prior to the arrival of modern technology, Persian was commonly written in two primary scripts: Nastaliq and Naskh. Rules for the scripts were passed on by word of mouth from masters to students. Thus, there were many styles of writing, whose popularity rested on the reputation of the practicing master. Among the rules were proper ways of generating combinations of letters (much like the "fi" & "ffi" combinations in English calligraphy). Because the Naskh script is more readily adaptable to modern technology, including to computer printers and displays, it has become more popular and has forked into many varieties in recent decades.

Nevertheless, Nastaliq holds a special place in the hearts and minds of Persian-speaking communities. The fanciest books of poetry are still produced in Nastaliq, and some printed flyers use Nastaliq for main headings to embellish and attract attention. Some progress has been made in producing the Nastaliq script automatically, and the results are encouraging. The Web site NastaliqOnline.ir allows its users to produce Nastaliq and a variety of other decorative scripts by entering their desired text within an input box. An image of the generated text can then be copy-pasted into other documents.

One final point about the Persian script, before entering the discussion of the three transition periods: On and off, over the past several centuries, reformation of the Persian script, to "fix" its perceived shortcomings in connection with modernity, has been the subject of heated debates. My personal view is that technology must be adapted to cultural, environmental, and linguistic needs, and not the other way around. Fortunately, success in producing high-quality print and display output has quelled sporadic attempts at reforming the Persian script or changing the alphabet [8], in a manner similar to what was done in Turkey, to save the society from "backwardness."

3. The Transition to Printing Press

The printing press arrived in Iran some 400 years ago (see the timeline in Fig. 2). Shah Abbas I was introduced to Persian and Arabic fonts and decided that he wanted them for his country [9]. A printing press and associated fonts were sent to Isfahan in 1629, but there is no evidence that they were ever put to use. Over the following decades, printing was limited mostly to a few religious tomes.

Broader use of printing technology dates back to 300 years ago. The invention of the Stanhope hand-press in 1800 revolutionized the printing industry, because it was relatively small and easy to use. This device was brought to Tabriz by those who traveled to Europe and Russia, around 1816 [10], and to Isfahan and Tehran a few years later, leading to a flurry of activity in publishing a large variety of books.

A key challenge in Persian printing was the making of the blocks that held the letters and other symbols (Fig. 3). English, with its comparably sized letters and the space between them, was much easier for printing than Persian, which features letters of widely different widths/heights, connectivity of adjacent letters, minor variations in letter shapes involving small dots (imagine having the letter "i" with 1, 2, or 3 dots), and more curvy letters.

Year	Events Affecting the Development of Persian Script
1600	
-	Printing press arrives in Iran; little/no use early on
-	Armenian press established in Jolfa, Isfahan
-	
-	
1700	
-	|
-	| Limited print runs; mostly on poetry and religion
-	|
-	Persian books published in Calcutta
1800	
-	First Stanhope hand-press arrives; printing spreads
-	Presses open in multiple cities; use of lithography
-	Technical books appear; newspapers flourish
-	
1900	First typewriter arrives in Iran
-	
-	Typewriters begin to be used widely
-	Electric typewriters, Linotype, and computers arrive
-	Standards for information code and keyboard layout
2000	Use of personal computers broadens
-	Computer-software and mobile-app industries thrive

Fig. 2. Rough timeline of key events and transitions in the history of adapting the Persian script to modern technology [9].

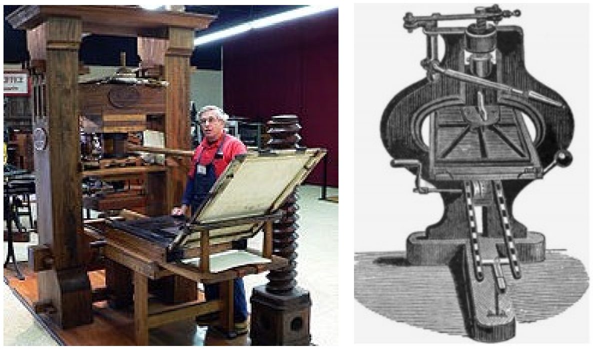


Fig. 3. Re-creation of Gutenberg's press at the International Printing Museum in Carson, California, USA (image: Wikipedia) and the Stanhope hand-press, introduced in 1800 [10].

The first order of business was to make the Persian script horizontally partitionable into letters that could then be juxtaposed to form the desired text. Pre-printing-press Persian script was not horizontally decomposable, as letters tended to mount each other vertically and overlap horizontally (bottom of Fig. 4). The modified form required some compromises in aesthetics, according to the prevailing tastes at the time (top-right of Fig. 4), which proved rather insignificant in retrospect.

Once conceptual changes were made, typographers got busy producing letters, letter combinations, and symbols for Persian printing (Fig. 5). We are now so used to the print-friendly Persian script that the pre-printing-press variants may look quaint to us!

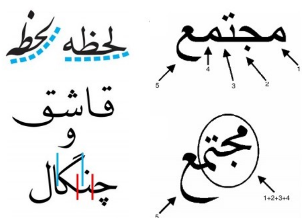


Fig. 4. For printing with movable type, the Persian script had to be made horizontally decomposable (uncredited Web images).

Fig. 5. Early Persian or Arabic metal fonts in the compartments of a typesetter's tray (uncredited Web image)

Fig. 6. Features of Persian script that make its printing difficult also create challenges in automatic text recognition [11].

The variable sizes and spacings of Persian letters also created manufacturing headaches for the font and difficulties for typesetters, who needed to handle blocks of widely different sizes. Interestingly, the features that make typesetting of Persian texts difficult are the same ones that make their automatic recognition challenging. As shown in Fig. 6, these include connectivity (a), error-causing minor differences (b), significant width variations (c), horizontal overlaps (d), and vertical overlaps (e).

Eventually, font designers succeeded in rendering the Persian alphabet with four shapes for each letter, in lieu of the nearly unlimited variations in calligraphic writing, where letters morph in shape, depending on the preceding and following letters (and sometimes, according to an even broader context). Still, with 4 variations for each letter, the number of different blocks needed was more than twice that of Latin-based scripts, the latter requiring a total of only 52 lowercase/uppercase letters. This made the utilization of typeface variations (boldface, italics, and the like) much more challenging.

Linotype, a hot-metal typesetting system invented by Ottmar Mergenthaler for casting an entire line of text via keyboard data entry, arrived in Iran in the 1950s, transforming and somewhat easing the typesetting problem for daily newspapers [12]. Contemporary Persian print output is now vastly improved (Fig. 7).

Fig. 7. Contemporary Persian newspaper print scripts. (Credit: *The Atlantic* Web site; Atta Kenare / Getty Images).

4. The Transition to Typewriting

Typewriters arrived in Iran around 120 years ago (Fig. 8), but much like the printing press, their use did not catch on right away. By the 1950s, many Western office-machine companies had entered Iran's market. Again, peculiarities of the Persian script created adaptation challenges.

Direct adoption of print fonts was impossible, given that with 32 letters, each having four variants, too many keys would be required. For most Persian letters, however, the initial and middle forms, and the solo and end forms, are sufficiently similar to allow combining, with no great harm to the resulting script's readability and aesthetic quality. Of course, early typewriters, all using fixed-width symbols, were ill-suited to the Persian script, with its highly-variable letter widths. It would be many years before variable-width symbols improved the Persian typewritten script quality substantially.

For example, the letters "meem" (م) and "beh" (ب) aren't too damaged by having two forms in lieu of four (Fig. 9). The same holds for "heh" (ه), at the left edge of Fig. 9, with slightly more distortion. The letters "ein" (ع) and "ghein" (غ) are the only exceptions needing all four variations (see the top-left of Fig. 9).

One of the highest-quality fonts for typewriters was offered by IBM in its Selectric line, which used a golf-ball print mechanism (right panels of Figs. 8 and 9). The golf-ball was easily removable for replacement with another golf-ball bearing a different font or alphabet (italic, symbol, etc.), making it easy to compose technical manuscripts involving multiple typefaces and equations. Even multiple languages could easily be incorporated in the same document. I used such a typewriter to produce my first textbook, *Computer Appreciation* [13], sample pages of which appear in Fig. 10.

Fig. 8. Mozaffar al-Din Shah's custom-made typewriter, ca. 1900 (Golestan Palace Museum, Tehran) and a later-model IBM Selectric with golf-ball printing mechanism, ca. 1975 (IBM).

Fig. 9. The four shapes of Persian letters and their reduction to two shapes in most cases (left; uncredited Web image) and IBM's Persian golf-ball print mechanism (personal photo).

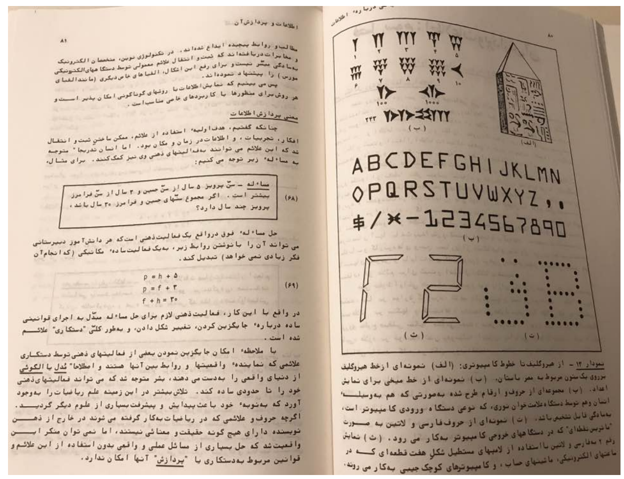


Fig. 10. Pages of the author's book *Computer Appreciation* [13] which he personally created on an IBM Selectric (Fig. 8, right) with a Persian golf-ball print mechanism (Fig. 9, right).

A common approach to building a Persian keyboard was to take an existing Arabic keyboard and add to it the four Persian-specific letters at arbitrary spots, giving rise to a multiplicity of layouts and making it difficult for typists to move between different typewriters. A standard Persian typewriter keyboard layout was thus devised [14]. Years later, standardization was taken up in connection with computer keyboards, creating the "Zood-Gozar" (زود گذر) layout [15], so named because of the sequence of letters at the very bottom row of Fig. 11, similar to the naming of the QWERTY keyboard. However, neither the keyboard layout nor the accompanying data interchange code [16] was adopted, given the pre-/post-revolutionary chaos.

Fig. 11. Unified Persian keyboard layout, a proposed standard for computers, typewriters, and other data-entry systems [15].

Intelligent typewriters soon arrived on the scene. First came word-processors that could store a line of text, thus allowing back-spacing to correct errors by striking the printing hammer on a white ribbon that would overwrite what was previously printed in a given position. This easy erasure mechanism is what allowed a non-professional typist like me to consider self-producing an entire book; cut-and-paste was, of course, still necessary for making larger corrections or moving paragraphs around.

The ultimate in intelligent typewriters, dubbed “word processors,” allowed the use of a single key for each letter, with a built-in algorithm deciding which variant of the letter to print. This required a one-symbol delay in printing, as the shape of each letter could depend on the letter that followed it. As an example, to print the word “kamtar” (کمتر), first the letter “kāf” (ك) would be entered. That letter would then be transformed from the solo/end variant to initial-middle form (ک), once the connectable letter “meem” (م) follows. This process continues until a space or line-break is encountered.

Interestingly, I cannot enter on my Microsoft Word program the initial/middle variant of “kāf” in isolation, as it is automatically converted to the solo/end variant. Thus, in the preceding paragraph, I was forced to connect something to “kāf” and then change the color of that letter to white, in order to make it disappear!

5. The Transition to Computer Printing

True word-processing and desktop publishing arrived in Iran in the 1980s [17], a few years after the worldwide personal-computer revolution. Prior to that, we produced Persian-script output on bulky line-printers and other kinds of printer devices connected to giant mainframes running in air-conditioned rooms of our computer centers, and, in later years, to mini- and micro-computers in our departmental and personal research labs.

One of the earliest computer printer technologies was the drum printer (Fig. 12, left). The rotating drum had one band of letters and symbols for each of the (typically 132) print positions. With the drum rotating at high speed, every letter/symbol would eventually be aligned with the print position, at which time, a hammer would strike on the paper and print ribbon, causing an impression of the raised symbol to be formed on the paper. A complete line was printed after one full revolution of the drum.

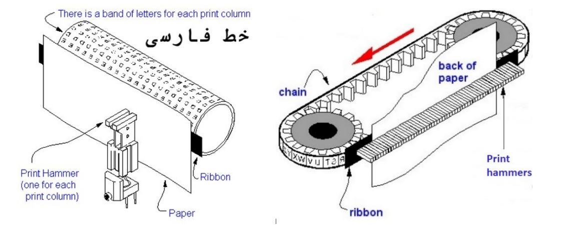


Fig. 12. Print mechanisms in early drum and chain printers (credit: *PC Magazine* Encyclopedia).

Drum printers were bulky and noisy, but, more importantly, were ill-suited to the production of legible Persian script. The separation of the bands of symbols on the drum and the spacing between adjacent hammers led to the appearance of white space between supposedly connected letters (Fig. 12, top-left). This space, combined with up- and down-shifting of symbols due to imprecision in the timing of hammer strikes, led to additional quality problems. The Latin script remains legible if adjacent letters are slightly up- or down-shifted, but the Persian script is much more sensitive to misalignment.

The problem with the bulk of drum printers was mitigated with chain (Fig. 12, right) and daisy-wheel printers, but print quality did not improve much, if at all. All three mechanisms suffered from smudging due to high-speed hammer strikes. Thus, letters appeared to be fuzzy, which, ironically, helped with filling the undesirable inter-symbol gaps, but it created additional legibility problems for similar-looking Persian letters.

Several other printing technologies came and went, until improvements in dot-matrix printing made all other methods obsolete. Early dot-matrix printers had a column of 7 pins that made contact with a ribbon to form small black dots on paper (Fig. 13, left). Then, either the needles moved to the next print column or the paper moved in the reverse direction, thereby forming symbols via printing 5 or more columns and continuing on until a complete line of text was formed.

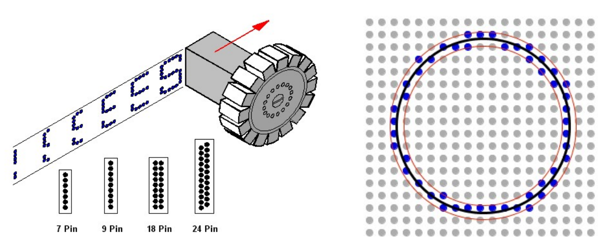


Fig. 13. Early dot-matrix print mechanism with a column of pins (left; credit: *PC Magazine* Encyclopedia) and the versatility of dot-matrix printing for producing images, in addition to text.

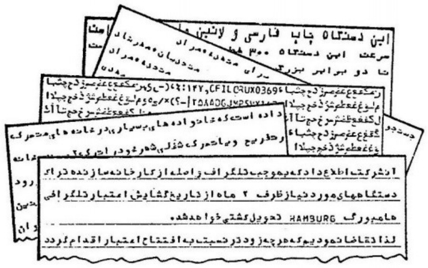


Fig. 14. Examples of Persian scripts produced by line printers and very early dot-matrix printers in the 1970s. [13]

Early dot-matrix printers, though convenient and economical, did not improve the quality of computer-generated Persian scripts, due to the matrix used being too small. In fact, there was a noticeable drop in print quality at first (Fig. 14). As matrix sizes grew and the dots were placed closer and closer to each other, the quality improved accordingly. We faced two categories of R&D problems in those days. First, given a dot-matrix size, how should the Persian letters and digits be formed for an optimal combination of legibility and aesthetic quality? Second, for a desirable level of legibility and aesthetics, what is the minimum required dot-matrix size?

To answer the first question, we would fill out matrices with letter designs and assemble them into lines (at first manually and later using a computer program) to check the script quality (Fig. 15, left). We then repeated the process with different matrix sizes to see the trade-offs. From these studies, we drew two key conclusions in connection with the second question.

First, for low-cost applications in which we cannot afford to use large dot-matrices, a lower bound of 9-by-9/2 dot-matrix size was established, below which legibility and quality become unacceptable. The simulation results for fonts in 7-by-5, 7-by-9/2, and 9-by-9/2 are depicted in Fig. 15, right. A matrix dimension $m/2$ implies the presence of m rows/columns of dots in skewed format, so that the physical dimension of the matrix is roughly $m/2$, despite the fact that there are m elements. This kind of skewed arrangement helps with generating fonts of higher quality, when the letters have curved or slanted strokes.

Second, we used the results from a Persian printed-text automatic recognition study to conclude that a "pen-width" of 4 is adequate for a legible and aesthetically pleasing script output (Fig. 16, left), although, of course, greater resolution can only help (Fig. 16, right).

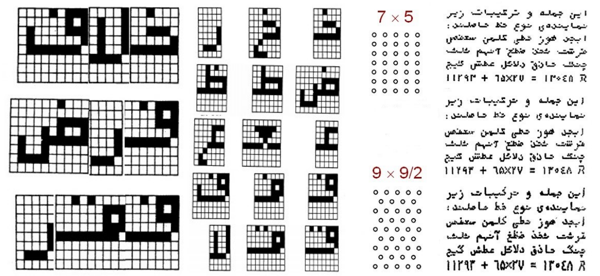


Fig. 15. Illustrating the design of dot-matrix fonts and juxtaposition of letters to check on the quality of the resulting script (left) and results of a study to establish a lower bound on the size of the dot-matrix for producing Persian script [18].

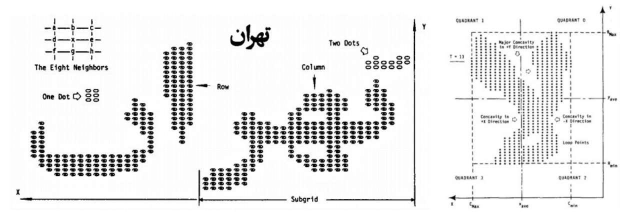


Fig. 16. Decomposition of connected Persian text into letters and recognizing the letters or composite forms [11].

In modern computer applications, a variety of Persian fonts are available to us. Legibility has improved significantly, but aesthetic quality is still lacking in some cases. In order to make small point sizes feasible, certain features of Persian letters must be exaggerated, so that details are not lost when font sizes are adjusted downward or when images are resized (as in fitting a map on the small screen of a mobile device). Some examples based on the Arial font appear in Fig. 17.

For actual modern computer-generated Persian scripts, I have chosen samples from Microsoft Word (Fig. 18). The samples show both high legibility/quality and problem areas (such as inordinately small dots for Tahoma).

Fig. 17. Illustrating the quality of Persian script using the Arial font of different sizes (top) and the effects of font-size adjustment and image resizing on readability of the resulting text.

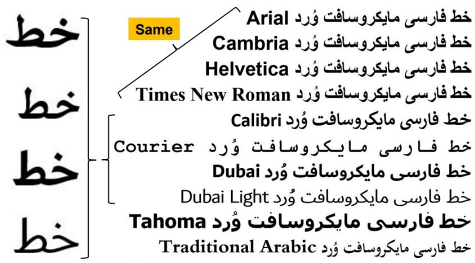


Fig. 18. Examples of modern Persian text output produced by Microsoft Word and the resulting script quality [1-2].

It appears that Calibri and Dubai fonts provide the best combination of legibility and aesthetic quality. The fixed-width Courier sample near the middle of Fig. 18 highlights the fact that fixed-width fonts produce even poorer-quality Persian text than is the case for Latin.

6. Digital Display Technologies

Displays used the dot-matrix approach much earlier than printers. CRT displays, in which an electron beam scans various “rows” on the screen, turning the beam on and off to produce a light or dark point on the screen’s coating, constitute a form of dot-matrix scheme. Before modern LCD or LED displays made the use of dot-matrix method for display universal, stadium scoreboards and airport announcement boards used a primitive form of dot-matrix display formed by an array of light bulbs.

For completeness of this historical perspective, I present a brief account of efforts to build Persian line-segment displays for calculators and other low-cost devices. The designs and simulated outputs are depicted in Fig. 19. Peculiarities of the Persian script made the designs of such displays a major challenge. We established that 7 segments would be barely enough for displaying Persian digits and that a minimum of 18 segments would be required for a Persian script that is readable (with some effort). Such displays became obsolete before the project moved to the production stage.

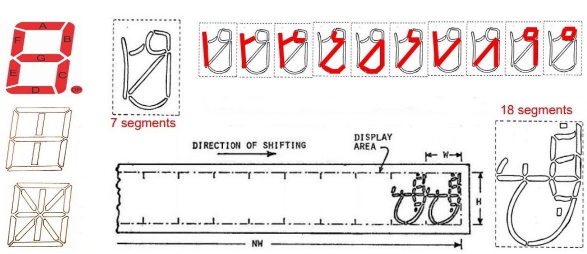


Fig. 19. Line-segment displays for Latin-based alphabets (left) and corresponding designs for Persian digits (top) and letters [1].

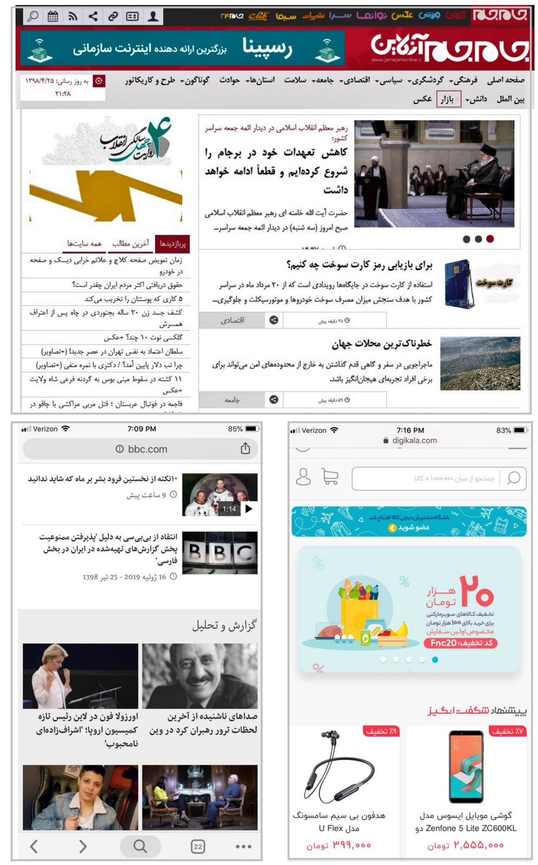


Fig. 20. Persian text displayed on Jam-e Jam news site of the government-run Islamic Republic of Iran Broadcasting system (top; laptop screen capture on July 16, 2019, 10:30 AM PDT) along with the BBC Persian news site and Digikala e-commerce site on a smartphone (bottom; captured the same afternoon).

Dot-matrix display methods are now producing Persian scripts that are comparable in quality to those of our best printers. The transition from CRTs to LCD, LED, and other modern display technologies has removed the flicker problem, the effect of low refresh rate which is particularly significant on CRT displays. Even though modern screens have a much larger number of dots, increases in processing rate and clock speed has made it less likely to have an inadequate refresh rate.

Examples of Persian scripts on modern displays, both spacious desktop/laptop screens and smaller screens of personal electronic devices, appear in Fig. 20. Web sites generally format their contents differently, depending on whether they are viewed on a big screen or a small screen, so that legibility does not become an issue even on the smallest device screens. It is however true that when such screens are viewed in bright environments, such as well-lit offices or outdoors, legibility may suffer.

7. Conclusion and Future Work

Today, technological tools for producing legible and aesthetically pleasing Persian script are widely available. So, whatever problems still remain are algorithmic and software-based in nature. Put another way, whereas until a couple of decades ago, computer typefaces had to be designed with an eye toward capabilities and limitations of printing and display devices, we can now return to typeface design by artists, with only aesthetics and readability in mind. Any typeface can now be mapped to suitably large dot-matrices to produce high-quality and easily-readable Persian script.

We now have reasonably good tools for generating and editing Persian texts. Among them are TeX systems for Arabic [19] and Persian [20], as well as many other text-processing systems based on Unicode [21]. Some popular programming languages also have built-in support for Persian text processing and I/O [22].

What remains to be done are systematic studies of trade-offs between Persian script legibility [23] and aesthetic quality and devising methods for taking care of formatting issues, particularly when bilingual text is involved. Use of crowdsourcing may help with solving the first problem. The second problem has persisted through many attempted solutions over several decades. It is still the case that when, for example, a Persian word is entered within an English text, or vice versa, the text may be garbled depending on the location of the alien word in the formatted line (e.g., close to a line break). An integrated, easy-to-use bilingual keyboard and improved optical character recognition would be important first steps in solving the remaining text-input problem.

References

[1] B. Parhami, "Computers and the Challenges of Writing in Persian: A Personal History Spanning Five Decades," being prepared for publication. (English version of [2])

[2] B. Parhami, "Computers and Challenges of Writing in Persian" (in Persian), *Iran Namag*, Vol. 4, No. 2, Summer 2019, to appear. (Persian version of [1])

[3] B. Parhami and F. Mavaddat, "Computers and the Farsi Language: A Survey of Problem Areas," *Information Processing 77* (Proc. IFIP World Congress), North Holland, 1977, pp. 673-676.

[4] B. Parhami, "On the Use of Farsi and Arabic Languages in Computer-Based Information Systems," *Proc. Symp. Linguistic Implications of Computer-Based Information Systems*, New Delhi, India, November 1978, pp. 1-15.

[5] B. Parhami, "Impact of Farsi Language on Computing in Iran," *Mideast Computer*, Vol. 1, No. 1, pp. 6-7, 1978.

[6] B. Parhami, "Language-Dependent Considerations for Computer Applications in Farsi and Arabic Speaking Countries," *System Approach for Development* (Proc. IFAC Conf.), North-Holland, 1981, pp. 507-513.

[7] G. Lazard, "The Rise of the New Persian Language," *The Cambridge History of Iran*, Vol. 4 (Period from the Arab Invasion to the Saljuqs), 2008, pp. 566-594.

[8] M. Borjian and H. Borjian, "Plights of Persian in the Modernization Era," *Handbook of Language and Ethnic Identity: The Success-Failure Continuum in Language and Ethnic Identity Efforts*, Vol. 2, pp. 254-267, 2011.

[9] W. M. Floor, "Čāp," *Encyclopedia Iranica*, I/7, pp. 760-764.

[10] N. Green, "Persian Print and the Stanhope: Industrialization, Evangelicalism, and the Birth of Printing in Early Qajar Iran," *Comparative Studies of South Asia, Africa, and the Middle East*, Vol. 30, No. 3, 2010, pp. 473-490.

[11] B. Parhami and M. Taraghi, "Automatic Recognition of Printed Farsi Texts," *Pattern Recognition*, Vol. 14, Nos. 1-6, pp. 395-403, 1981.

[12] T. Nemeth, *Arabic Type-Making in the Machine Age: The Influence of Technology on the Form of Arabic Type, 1908-1993*, Brill, Leiden, 2017, p. 288.

[13] B. Parhami, *Computer Appreciation* (in Persian), Tehran, Tolou'e Azadi, 1984.

[14] Institute of Standards and Industrial Research of Iran, *Character Arrangement on Keyboards of Persian Typewriters* (in Persian), ISIRI 820, 1976.

[15] B. Parhami, "Standard Farsi Information Interchange Code and Keyboard Layout: A Unified Proposal," *J. Institution of Electrical and Telecommunications Engineers*, Vol. 30, No. 6, pp. 179-183, 1984.

[16] Iran Plan and Budget Organization, *Final Proposal for the Iranian National Standard Information Code* (INSIC), Persian and English versions, 1980.

[17] M. Sanati, "My Recollections of Desktop Publishing" (in Persian), *Computer Report*, Vol. 40, No. 239, pp. 53-60, Fall 2018.

[18] B. Parhami, "On Lower Bounds for the Dimensions of Dot-Matrix Characters to Represent Farsi and Arabic Scripts," *Proc. 1st Annual CSI Computer Conf.*, Tehran, Iran, December 1995, pp. 125-130.

[19] K. Lagally, "ArabTeX, a System for Typesetting Arabic," *Proc. 3rd Int'l Conf. Multi-lingual Computing: Arabic and Roman Script*, Vol. 9, No. 1, 1992.

[20] B. Esfahbod and R. Pournader, "FarsiTeX and the Iranian TeX Community," *TUGboat*, Vol. 23, No. 1, pp. 41-45, 2002: https://tug.org/TUGboat/tb23-1/farsitex.pdf

[21] Unicode.org, "About the Unicode Standard," on-line resource page with pertinent links, accessed on July 16, 2019: https://unicode.org/standard/standard.html

[22] Python.org, "Links to Python Information in Persian/Iranian/Farsi," On-line resource page, accessed on July 16, 2019: https://wiki.python.org/moin/PersianLanguage

[23] N. Chahine, "Reading Arabic: Legibility Studies for the Arabic Script," Doctoral Thesis, Leiden University, 2012.

The unreasonable effectiveness of pattern generation

Petr Sojka, Ondřej Sojka

Abstract

Languages are constantly evolving, and so are their hyphenation rules and needs. The effectiveness and utility of TeX's hyphenation have been proven by its usage in almost all typesetting systems in use today. The current Czech hyphenation patterns were generated in 1995, and no hyphenated word database was freely available.

We have developed a new Czech word database and have used the `patgen` program to generate new effective Czech hyphenation patterns efficiently and evaluated their generalization qualities. We have achieved full coverage on the training dataset of 3,000,000 words and developed a validation procedure of new patterns for Czech based on the testing database of 105,000 words approved by the Czech Academy of Science linguists.

Our pattern generation case study exemplifies a practical solution to the widespread dictionary problem. The study has proved the versatility, effectiveness, and extensibility of Liang's approach to hyphenation developed for TeX. The unreasonable effectiveness of pattern technology has led to applications that are and will be used, even more widely now, nearly 40 years after its inception.

> ...the best approach appears to be to embrace the complexity of the domain and address it by harnessing the power of data: if other humans engage in the tasks and generate large amounts of unlabeled, noisy data, new algorithms can be used to build high-quality models from the data. (Peter Norvig, [7])

1 Introduction

In their famous essays, Wigner [19], Hamming [1] and Norvig [7] consider mathematical and data-driven approaches to be miraculously, unreasonably effective. One of the very first mathematically founded approaches that harnessed the power of data was Franklin Liang's language-independent solution for TeX's hyphenation algorithm [6] and his program `patgen` for a generation of hyphenation patterns from a word list.

Dictionary problem The task at hand was a *dictionary problem.* A dictionary is a database of records; in each record, we distinguish the key part (the word) and the data part (its division). Given an already hyphenated word list of a language, a set of *patterns* is magically generated. Hyphenation patterns are much smaller than the original word list and typically encode almost all hyphenation points in the input list without mistakes. Liang's pattern approach thus could be viewed as an efficient lossy, ideally lossless, *compression* of the hyphenated dictionary with a compression ratio of several orders of magnitude.

It has been proved [16, chapter 2] that the optimization problem of exact lossless pattern minimization is non-polynomial by reduction to the minimum set cover problem.

Generated patterns have minimal length, e.g., shortest context possible, which results in their *generalization* properties. Patterns could hyphenate words not seen during learning: yet another miracle of the generated patterns.

Pattern preparation In the 36 years of `patgen` use, there have been hundreds of hyphenation patterns created, either by hand or *generated* by the program `patgen`, or by the combination of both methods [8]. The advantage of *pattern generation* is that one can fine-tune pattern qualities for specific usage. Having an open-source and maintained word list adds another layer of flexibility and usability to the deployment of patterns. This approach is already set up for German variants and spellings [5] and was an inspiration for doing the same for the Czech language.

In this paper, we report on the development of the new Czech word list with a free license and complementary sets of hyphenation patterns. We describe the iterative process of initial word list preparation, word form collection, estimation of pattern generation parameters, and novel applications of the technology.

> Hyphenation is neither anarchy nor the sole province of pedants and pedagogues. Used in moderation, it can make a printed page more visually pleasing. If used indiscriminately, it can have the opposite effect, either putting the reader off or causing unnecessary distraction. (Major Keary)

2 Initial word list preparation

As a rule of thumb, the development of a large new hyphenated word list starts with a small dataset. The experience and outputs from this initial phase, e.g., hyphenation patterns, are then applied to the larger and larger lists.

Bootstrapping idea As word lists of a well-established language are sizeable, and manual creation of a huge hyphenated word list is tedious work, we used the bootstrapping technique. We illustrate the process of initial word list preparation in the diagram in Figure 1 on the following page. We have

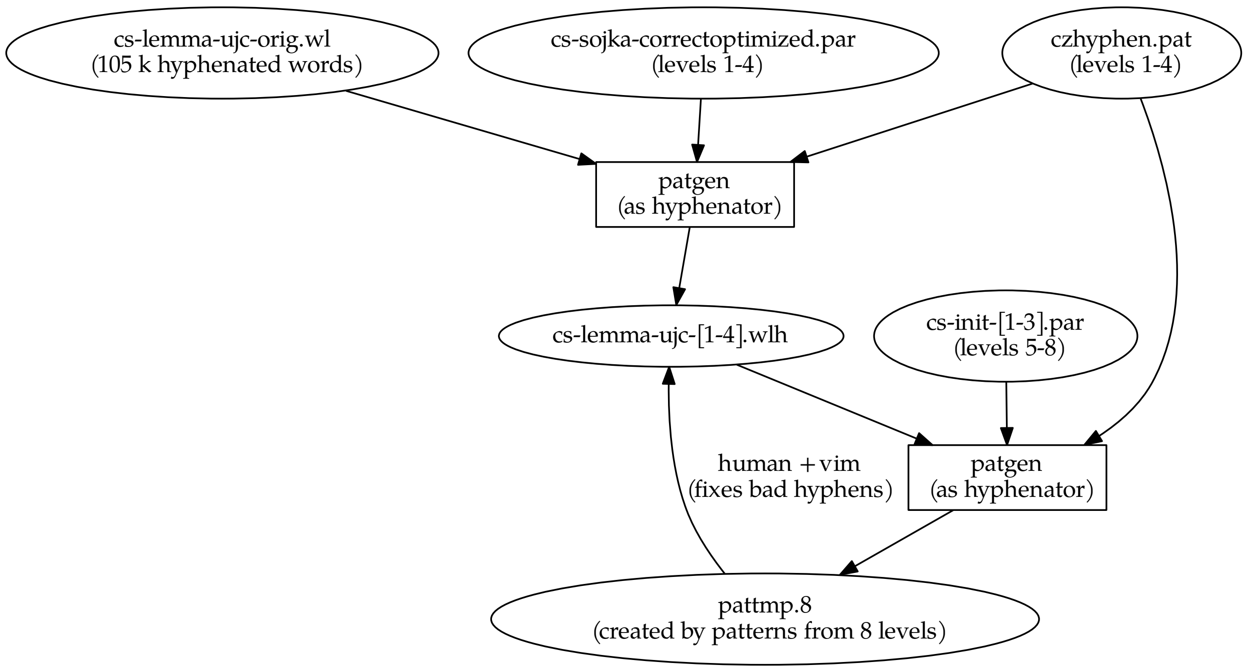


Figure 1: Life cycle of initial word list preparation, illustrated with the development of 105 k Czech consistently hyphenated words. `czhyphen.pat` represents the original Czech hyphenation patterns from [17] and `cs-sojka-correctoptimized.par` are correct optimized `patgen` parameters from the same paper. `cs-init-[1-3].par` are custom parameters that trade off bad hyphens (which have to be manually checked) for missed hyphens. Information on which hyphenations `patgen` missed, and where it wrongly inserted a hyphen is sourced from `pattmp`.

obtained a hyphenated word list with 105,244 words from the Czech Academy of Sciences, Institute of the Czech Language (ÚJČ). Upon closer inspection, we discovered many problems with the data, probably stemming from the facts that it has been crafted by multiple linguists and over many years. The few hyphenation rules [2] that are in the Czech language are not applied consistently. The borderline cases were typically between syllabic (ro-zum) and etymological variants (roz-um) of hyphenation, or the way to handle words borrowed from German or English into Czech. There are sporadic examples of words where correct syllabification depends on the semantics of the word: *narval* and *oblít* are two examples of them in Czech. These are preferably not to be hyphenated, to stay on the safe side.

It is impractical to try to manually find inconsistencies and systemic errors, even in a relatively short word list like this. We slightly modified and extended the process suggested in [15, page 242]: We used `patgen` and the current Czech patterns to hyphenate the word list and manually checked only the 25,813 words where the proposed hyphenation points differed from the official (were bad or missed), creating a new word list `cs-lemma-ujc-1.wlh` [13] in the process.

However, we are erroneous humans making mistakes. To find these, we have used `patgen` to generate the four additional levels of hyphenation patterns on top of the current patterns from the checked word list. We have also adjusted the parameters (see `cs-init-[1-3].par` [13]) used for generation of the four additional levels to trade off bad hyphens (which have to be manually checked) for missed ones. We have then used these patterns, with eight levels in total, to hyphenate the checked word list and manually rechecked the wrongly hyphenated points (dots in `patgen` output), with missed hyphenation points (implicitly marked as the hyphen sign in hyphenated word list). We have repeated this process three times, iterating on `cs-lemma-ujc-[2-4].wlh`. Word list number four is used for the generation of bootstrapping patterns and final pattern validation.

3 Word list preparation and design

Any live language continually changes, and Czech is no exception. Many new Czech words now come from other languages, mostly from English. It presents a challenge for the patterns; they must not only correctly hyphenate Czech words according to Czech syllabic boundaries, but foreign words must be hyphenated correctly too, according to their new Czech syllabic pronunciation [14]. To have the patterns keep up with language evolution, we must maintain not only the patterns but also a hyphenation word list. In this section, we detail how we have built such a word list.

csTenTen corpus We have first obtained a word list with frequencies, generated from the Czech Web Corpus of TenTen family (csTenTen) [3]. We then filtered this word list to include only words that appear more than ten times in two crawls [18] made in years 2012 and 2017. We ended up with a word list containing 922,216 words, a non-negligible fraction of which are misspellings and jargon.

Word list cleanup We have then cleaned this word list by using the Czech morphological analyzer `majka` [12] to remove all words not known to it. We removed 370,291 typos, misspellings, and similar atypical lexemes and kept only 551,925 frequently occurring valid words in the dataset.

Word list expansion The morphological analyzer `majka` [12] also allows us to expand words into all their inflected forms. We chose not to use the expansion feature of `majka` because the word list would grow to 3,779,379 (almost a fourfold increase) and csTenTen already contains most of the commonly used types of inflections. It would also distort which hyphenation `patgen` gives the most weight to. We tried supplying logarithms of word frequencies from csTenTen to the word list, so more weight could be given to patterns that cover the most common words. It did not significantly improve validation scores in our case, as one can see in Table 2 on page 191. We think that this is partly because `patgen` is limited to one digit of frequency per word and partly because the validation score (computed from error rate on `ujc` word list) does not capture real-world usage.

We expanded the word list with `majka` by adding 54,569 lemmas (base forms) that were present in the word list, but not in their base form. It increased the word list size to 606,494 words.

We list the word list statistics that we used for pattern generation in Figure 2.

shortcut	word list description	count
`ujc`	checked word list for validation	105,244
`all`	all frequent word forms from web known to `majka` plus all lemmas known to `majka`	606,494
`allflex`	previous plus all word forms generated by majka	2,100,581
`allflexjargon`	previous plus all non-standard and jargon word forms	3,779,379
`biggest`	tokens that are present in the csTenTen more than 10 times	3,918,054

Figure 2: Czech word lists' shortcut names and statistics

Maintenance The German *wortliste* [5] project served as inspiration for our open word list format, detailed in the `README.md` [13].

> One must regard the hyphen as a blemish to be avoided wherever possible. (Winston Churchill)

4 Bootstrapping — iterative development of hyphens in the big word list

It would be tedious to hyphenate such a big word list by hand manually, so we train patterns on a small list and apply them to the big word list, as illustrated in Figure 3 on the next page. Then, we train patterns on the (now hyphenated) big word list and have `patgen` show what it would have hyphenated differently. With this approach, we cherry-pick inconsistencies in the word list.

Since the big word list contains not only lemmas of words, but also characteristic inflections, we use regular expressions to add hyphens around them and fix inconsistencies. We keep iterating on this, as shown in Figure 3 on the following page, until the patterns, generated with `cs-init-[1-3].par` [13], achieve nearly perfect coverage.

The resulting patterns hyphenate according to the standard Czech hyphenation rule: hyphenation is allowed everywhere where it does not change the pronunciation of the word. Thanks to the effectiveness of pattern generation, this works not only in Czech words but also foreign (Latin, French, German, English) ones.

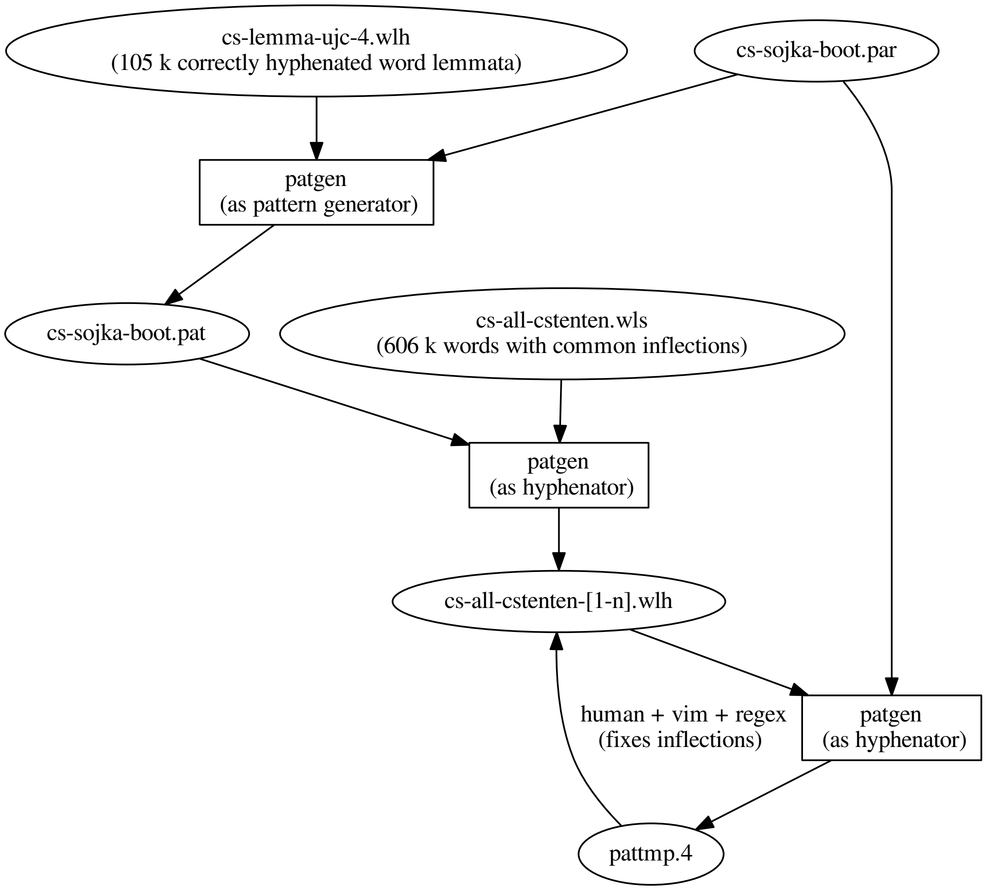


Figure 3: How we bootstrapped hyphenation of the big word list by training patterns (`cs-sojka-boot.pat`) on the small word list and applying them on the big one. `cs-sojka-boot.par` are `patgen` parameters that are designed to generate many patterns but still retain their generalization properties. `pattmp` highlights which hyphenation points in the source file the new pattern level missed, which were correctly covered and where they wrongly put a hyphen.

Hyphens, like cats, are capable of arousing tenderness or shudders. (Pamela Frankau)

5 Pattern generation

The last Czech hyphenation patterns were generated in 1995 [17], and are in use not only in TEX but also in other widespread typesetting systems. For conservative users, there is no strong incentive for change, because the error rate is relatively low (the first version of the validation set measured an error rate around 4%), and coverage is relatively high (the first version of the validation set measured around 7% missed hyphenation points).

Pattern generation from 3,000,000 words does not take hours as it did two decades ago, but seconds, even on commodity hardware, which allows for rapid development of “home-made” patterns.

We have developed a Python wrapper for `patgen` that we use in Jupyter notebooks. It allows rapid iteration, and easy sharing of results — see Table 1 on the next page and `demo.ipynb` [13].

Had Liang in 1983 had the same ease of changing `patgen` parameters, run it, and see the results in 60 seconds, he would inevitably have generated higher than 89% coverage while staying within the limit of 5,000 patterns [6, page 37].

It has also become common to use a validation dataset to ensure generalization abilities. Our usage of a validation dataset has proved useful. Table 2 shows that if we were to use the *correct optimized* parameters from [17] that have been in use for Czech, we would overfit the training dataset and perform *worse* than their *size optimized* counterparts. The

Table 1: Outputs from running `patgen` in our Jupyter notebook with two different parameter sets. The first parameter set is from the German Trennmuster project [5] and generates 7,291 patterns, 40 kB. The second one from [17] generates shorter and smaller patterns — 4,774 patterns, 25 kB.

Level	Patterns	Good	Bad	Missed	Lengths	Params
1	750	1,683,529	525,670	0	1 5	1 1 1
2	3,178	1,628,874	38	54,655	2 6	1 2 1
3	2,548	1,683,528	9,931	1	3 7	1 1 1
4	1,382	1,683,287	0	242	4 8	1 4 1
5	92	1,683,528	0	1	5 9	1 1 1
6	0	1,683,528	0	1	6 10	1 6 1
7	1	1,683,529	0	0	7 11	1 4 1

Level	Patterns	Good	Bad	Missed	Lengths	Params
1	1,608	1,655,968	131,481	27,561	1 3	1 5 1
2	1,562	1,651,840	2,533	31,689	1 3	1 5 1
3	2,102	1,683,528	2,584	1	2 5	1 3 1
4	166	1,683,135	6	394	2 5	1 3 1

Table 2: Effectiveness and effectivity of pattern generation on Czech word lists. Comparison of validation scores of patterns trained on various word list and parameter combinations.

Word list	Params	Good %	Bad %	Missed %	Size	Patterns	Time (s)
`all`	correctopt [17]	99.76	2.94	0.24	30 kB	5,593	58.13
	sizeopt [17]	98.95	2.80	1.05	19 kB	3,816	59.46
	german [5]	99.74	2.21	0.26	51 kB	8,991	201.9
weighted `all`	correctopt [17]	99.76	2.94	0.24	30 kB	5,590	59.23
	sizeopt [17]	98.95	2.80	1.05	20 kB	3,821	58.74
	german [5]	99.74	2.21	0.26	51 kB	8,978	207.35
`allflex`	correctopt [17]	99.46	4.02	0.54	28 kB	5,387	212.55
	sizeopt [17]	99.26	3.72	0.74	29 kB	5,537	212.59
	german [5]	99.42	3.35	0.58	49 kB	8,663	1,035.16
`allflexjargon`	correctopt [17]	99.47	4.08	0.53	29 kB	5,612	365.96
	sizeopt [17]	99.31	3.78	0.69	31 kB	5,938	369.92
	german [5]	99.43	3.36	0.57	53 kB	9,308	1,786.4

validation word list has to be carefully checked with linguists from UJČ for consistency to minimize the generalization error. Most of the current errors stem from foreign words used in the Czech texts.

When the validation word list is added to training, then patterns could be developed to serve as a lossless compression of word list dataset, thus maximize the effectiveness of pattern technology.

Life is the hyphen between matter and spirit.
(Augustus William Hare)

6 The unreasonable effectiveness

We were able to solve the dictionary problem for Czech hyphenation effectively.

Space effectiveness From 3,000,000+ hyphenated words stored in approximately 30,000,000 bytes we have produced patterns of size 30,000 bytes, achieving roughly 1000× space *lossless* compression.

Time effectiveness Using the trie data structure for patterns makes the time complexity of accessing the record related to the word, e.g., hyphenation point, in very low *constant* time. The constant is related to the depth of the pattern trie data structure, e.g., 5 or 6 in the case of Czech. If the entire pattern trie resides in RAM, the time for finding the patterns for a word is on the scale of tens, at most hundreds, of single processor instructions. Word hyphenation throughput is then about 1,000,000 words per second on a modern CPU.

Optimality Even though finding exact space and time-optimal solutions is not feasible, finding an approximate solution close to optimum is possible. Heuristics and insight expressed above, together with interactive fine-tuning of `patgen` parameter options, in our case on a Jupyter notebook, allows for rapid pattern development.

Automation A close-to-optimal solution to the dictionary problem could be useful not only for Czech hyphenation, but for all other languages [8, 9], and more generally, for other instances of the dictionary problem. Developing heuristics for thresholding of `patgen` pattern generation parameters could be based on a statistical analysis of large input datasets. It could allow the deployment of presented approaches on a much broader problem set and scale. We believe that parameters could be approximated *automatically* from the statistics of the input data.

Pattern generation — in Wigner's words — "has proved accurate beyond all reasonable expectations". Let us paraphrase another one of his quotes:

> The miracle of the appropriateness of the language of ~~mathematics~~ *patterns* for the formulation of the laws of ~~physics~~ *data* is a wonderful gift which we neither understand nor deserve. We should be grateful for it and hope that it will remain valid in future research and that it will extend, for better or for worse, to our pleasure, even though perhaps also to our bafflement, to wide branches of learning.

> "We should stop acting as if our goal is to author extremely elegant theories, and instead embrace complexity and make use of the best ally we have: the unreasonable effectiveness of data." (Peter Norvig, [7])

7 Conclusion

We have developed a flexible open language–independent system [13] for hyphenation pattern generation. We have demonstrated the effectiveness of this system by updating the old Czech hyphenation patterns [17] and achieving record accuracy. We have also applied recent data and computer science advancements, like the usage of interactive Jupyter notebooks and a validation dataset to prevent overfitting, to the more than three decades old problem of pattern generation.

Future work

Word lists for other languages The logical next steps will be applying developed techniques for different languages: for Slovak and virtually all others that do not yet have word list–based hyphenation patterns, and a word list either in Sketch Engine or elsewhere is available.

Stratification Pattern generation could be further sped up by several techniques, such as stratification of word lists on the level of input, or on the level of counting pro and con examples to include a new pattern or not.

Pattern-encoded spellchecker We have a big dictionary of frequent spelling errors from the csTenTen word list. Nothing prevents us from encoding these into specific patterns or pattern layers with extra levels and use that information during typesetting, e.g., to typeset those words with red underlining in LuaTEX. LuaTEX allows dynamic pattern loading and Lua programming that can enable the implementation of this feature, which people are used to having in editors.

Word segmentations Recent progress in machine-learned natural language processing and machine translation builds on subword representations and various types of semantically coherent sentence or word segmentations. As tokenization and segmentation are at the beginning of every natural language processing pipeline, there is a demand for effective and efficient universal segmentation [11]. New neural machine translation systems are capable of open-vocabulary translation by representing rare and unseen words as a sequence of subword units [10, Table 1]. Segmentation is crucial, especially for compositional languages like German, where there are many compounds (mostly out of vocabulary words) and for morphologically rich languages like Hebrew [20] or Arabic, that need to be segmented, represented, and translated.

Pattern-based learnable key memories Solutions to versions of the dictionary problem are a hot topic of leading-edge research to design memory data architectures like those used in machine learning of language [4]. Pattern-based memory network architectures could speed up language data access in huge memory neural networks considerably.

Multilingual hyphenation patterns Given that there are close languages with syllabic-based rules like Czech and Slovak, generating patterns from merged word lists is straightforward. It would save energy on low-resource devices like e-book readers by having them load fewer patterns at a time.

Acknowledgments The authors thank the TeX Users Group and CSTUG for financial support to present the project at TUG 2019. We owe our gratitude also to Vít Suchomel of Lexical Computing for word lists from Sketch Engine, to Pavel Šmerk, Frank Liang and Don Knuth for `majka`, `patgen` and TeX, respectively. Thanks go to Vít Novotný and Pavel Šmerk for valuable comments to the paper.

References

[1] R. W. Hamming. The unreasonable effectiveness of mathematics. *The American Mathematical Monthly* 87(2):81–90, 1980. `http://www.jstor.org/stable/2321982`

[2] Internetová jazyková příručka (Internet Language Reference Book). Institute of Czech language, Czech Academy of Sciences. `http://prirucka.ujc.cas.cz/?id=135`

[3] M. Jakubíček, A. Kilgarriff, et al. The TenTen Corpus Family. In *Proc. of 7th International Corpus Linguistics Conference (CL)*, pp. 125–127, Lancaster, July 2013.

[4] G. Lample, A. Sablayrolles, et al. Large memory layers with product keys, 2019. `https://arxiv.org/pdf/1907.05242`

[5] W. Lemberg. A database of German words with hyphenation information. `https://repo.or.cz/wortliste.git`

[6] F. M. Liang. *Word Hy-phen-a-tion by Com-put-er.* PhD thesis, Department of Computer Science, Stanford University, Aug. 1983. `tug.org/docs/liang`

[7] F. Pereira, P. Norvig, and A. Halevy. The unreasonable effectiveness of data. *IEEE Intelligent Systems* 24(02):8–12, Mar. 2009. `doi:10.1109/MIS.2009.36`

[8] A. Reutenauer and M. Miklavec. TeX hyphenation patterns. `tug.org/tex-hyphen`

[9] K. P. Scannell. Hyphenation patterns for minority languages. *TUGboat* 24(2):236–239, 2003. `tug.org/TUGboat/tb24-2/tb77scannell.pdf`

[10] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, Berlin, Germany, Aug. 2016. Association for Computational Linguistics. `doi:10.18653/v1/P16-1162`

[11] Y. Shao, C. Hardmeier, and J. Nivre. Universal word segmentation: Implementation and interpretation. *Transactions of the Association for Computational Linguistics* 6:421–435, 2018. `doi:10.1162/tacl_a_00033`

[12] P. Šmerk. Fast morphological analysis of Czech. In P. Sojka and A. Horák, eds., *Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2009*, pp. 13–16, Karlova Studánka, Czech Republic, Dec. 2009. Masaryk University. `http://nlp.fi.muni.cz/raslan/2009/`

[13] O. Sojka and P. Sojka. cshyphen repository. `https://github.com/tensojka/cshyphen`

[14] P. Sojka. Notes on compound word hyphenation in TeX. *TUGboat* 16(3):290–297, 1995. `tug.org/TUGboat/tb16-3/tb48soj2.pdf`

[15] P. Sojka. Hyphenation on demand. *TUGboat* 20(3):241–247, 1999. `tug.org/TUGboat/tb20-3/tb64sojka.pdf`

[16] P. Sojka. *Competing Patterns in Language Engineering and Computer Typesetting.* PhD thesis, Masaryk University, Brno, Jan. 2005.

[17] P. Sojka and P. Ševeček. Hyphenation in TeX — Quo Vadis? *TUGboat* 16(3):280–289, 1995. `tug.org/TUGboat/tb16-3/tb48soj1.pdf`

[18] V. Suchomel and J. Pomikálek. Efficient web crawling for large text corpora. In A. Kilgarriff and S. Sharoff, eds., *Proc. of the Seventh Web as Corpus Workshop (WAC)*, pp. 39–43, Lyon, 2012. `http://sigwac.org.uk/raw-attachment/wiki/WAC7/wac7-proc.pdf`

[19] E. P. Wigner. The Unreasonable Effectiveness of Mathematics in the Natural Sciences. Richard Courant Lecture in Mathematical Sciences delivered at New York University, May 11, 1959. *Communications on Pure and Applied Mathematics* 13(1):1–14, 1960. `doi:10.1002/cpa.3160130102`

[20] A. Zeldes. A characterwise windowed approach to Hebrew morphological segmentation. In *Proc. of the Fifteenth Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pp. 101–110, Brussels, Belgium, Oct. 2018. Association for Computational Linguistics. `doi:10.18653/v1/W18-5811`

◇ Petr Sojka
Faculty of Informatics, Masaryk University
Brno, Czech Republic and CSTUG
`sojka (at) fi dot muni dot cz`
`https://www.fi.muni.cz/usr/sojka/`

◇ Ondřej Sojka
CSTUG, Brno, Czech Republic
`ondrej.sojka (at) gmail dot com`

Improving Hangul to English translation for optical character recognition (OCR)

Emily Park, Jennifer Claudio

Abstract

Real-time translation of languages using camera inputs occasionally results in awkward failures. As a proposed method of assisting such tools for Korean (Hangul) to English translation, an optical assessment method was proposed to help translation algorithms first assess whether the Korean text has been written as English syllables in Korean or in true Korean vocabulary before producing translated phrases. Although the current approach was not viable, future work will implement feedback regarding methods to meaningfully handle the optical data received.

1 Introduction

1.1 Korean alphabetic syllabary

Hangul, the writing system of the Republic of Korea, currently uses an alphabet constituent of 14 consonants and 10 vowels. The Hangul alphabet is described as an alphabetic syllabary, meaning that although alphabet units consist of vowels and consonants working together to depict a sound, letter and syllable combinations have both a vertical and horizontal relationship. This relationship is in contrast to a language such as English, where each alphabetic letter has only a horizontal relationship with the ones that precede or follow it. In Korean, sets of syllables thus produce words which, to a non-Korean speaker, must be converted into semantic units.

Hangul has changed immensely over its history, including historic concern about class differences in the original Korean writing systems to inclusion of the modernized systems. Even more recently, the spread of *Konglish*, words derived from English but used in a Korean context, pose new issues and face new criticisms. Firstly, the linguistic divide between North and South Korea is further emphasized by the divergence of word choice or usage, and secondly transliterations require contextual relevance, as otherwise a homograph may be substituted by the reader. When read by a human, this context can easily be picked up through visual cues such as images associated with the text or with contiguous lines of text, however, an Optical Character Recognition (OCR) reader may only grasp sequential words and less context, hence leading to mistranslation of words.

1.2 Language conversion and accessibility

The relevant forms of language conversion for this situation are transliteration and translation. Transliteration provides a syllabic conversion using characters of another alphabet, whereas translation provides the meaning of a word in a different language.

To couple linguistic and physical accessibility, Optical Character Recognition (OCR) is widely used for recognizing text from scanned documents and converting them to editable data. One method of language-relevant OCR is through Google Translate, though many other programs and platforms exist. As many individuals who have used an OCR language conversion on a food menu may know, awkward translations can occur due to insufficient context for the reader or due to literal translations and loss of figurative speech. Furthermore, the adoption of Konglish presents a problem where a language learner or a speaker who has had less exposure to English may not recognize a word that is actually English that has been transliterated into Korean.

2 Goals

The goal of this project was to create a predictive method for text conversion as an alternative or supplement to sole reliance on counting database references. In doing so, such a predictive method would improve results for Korean language learners and older or more traditional speakers.

Currently, language translators rely on a database of known words, and many include common transliterations and borrowed words. The functionality of any OCR-based language translator therefore depends on the size and integrity of its associated database. While some translators have mentioned AI implementation based on word associations or probability calculations of word linkage, this aspect is beyond the scope of this project.

3 Methods

Fifty common words in English and Korean were both transliterated phonetically and translated across languages to assess preliminary data and feasibility testing. A script was written using Python to determine pixel area represented by the text and the area of its bounding box as determined by the outermost edges of a word's letters. Image samples were taken from different print media, specifically from newspapers, children's books, and advertisements. Each image was fed into the software five times to test reliability, then the ratios of text space to background space were tabulated and calculated. This was done to determine if the ratio retrieved from an OCR could inform the translator as to whether the word was true Korean or transliterated English.

The program imported the *cv2* (a.k.a. OpenCV) Python package for image recognition. The code first filtered the image into a pure black and white image. Subsequently, the height and width were calculated by detecting the number of pixels that comprised the word. The code looped over every pixel and finally printed the total number of black pixels in the filtered image. The gathered results of black pixels over the area of the bordered word fell between the ratios of 0.3 and 0.5.

Initially, the code used for bordering the text first recognized the words in the image and traced around the detected word to define the border. This was attempted with *pytesseract* (Python Tesseract), which was found unfeasible for recognizing words; this was thus adjusted to use the Google platform to recognize and translate their images.

Upon refining the script to find a bounding edge and verifying that the difference in area between foreground text and background could be determined, word samples were collected and processed.

4 Results and discussion

Although the code generated was able to perform calculations of text versus background, the ratios of foreground to background were not statistically different from each other across transliterated and translated words. This is attributed to the limited number of characters that comprise Hangul, of which fewer than five basic shapes (vertical sticks, horizontal sticks, circles, boxes, and huts). This shape limitation restricts the number of symbols that could be formed. An alternative method of informing a translation platform would be to assess the number of strokes in a word and the number of words that use a given consonant sound. The number of strokes could be viable because many transliterated words result in three syllables, despite being a single syllabic word when pronounced in native Korean.

A secondary issue with the input method included the necessary conversion into black and white. This text then needed to be manually uploaded into the translator, rather than performing in real time in tandem with the OCR itself. This method consequently defeats the purpose of pairing with an OCR.

As expected, font, typeface, and stylizations affected ratios, however, this was determined not to be a contributing factor to the inability to create predictive translation.

5 Conclusions

In the manner approached, using text to background ratios is *not* a viable method of implementing a predictive algorithm without context. Current AI methods used by translators exhibit fairly consistent performance.

6 Acknowledgments

Special thanks to the TeX Users Group and its associated community for their support during our attendance at the annual conference. Additional thanks to Govind and Ganesh Pimpale for their support with generating the Python code for OCR use.

◇ Emily Park, Jennifer Claudio
Oak Grove High School
Science Research Program
San Jose, CA

A glance at CJK support with XƎTEX and LuaTEX

Antoine Bossard

Abstract

From a typesetting point of view, the Chinese and Japanese writing systems are peculiar in that the characters are concatenated without using spaces to separate them or the meaning units (i.e., "words" in our occidental linguistic terminology) they form. And this is also true for sentences: although they are usually separated with punctuation marks such as periods, spaces remain unused. Conventional typesetting approaches, TEX in our case, thus need to be revised in order to support the languages of the CJK group: Chinese, Japanese and, to a lesser extent, Korean. While more or less complete solutions to this issue can be found, in this article we give and pedagogically discuss a minimalistic implementation of CJK support with the Unicode-capable XƎTEX and LuaTEX typesetting systems.

1 Introduction

The Chinese, Japanese and Korean writing systems are conventionally gathered under the CJK appellation. The Chinese writing system consists of the Chinese characters, which can be in simplified or traditional form, amongst other character variants [1]. The (modern) Japanese writing system is made of the Chinese characters and the kana characters. The Chinese and Japanese writing systems concatenate characters without ever separating them with spaces. The Korean writing system consists mainly of hangul characters, in principle together with the Chinese characters, but they are rarely used nowadays. Although modern Korean does separate words with spaces, traditionally, the Korean writing system does not (as an illustration, see, e.g., Sejong the Great's 15^{th} century manuscript Hunminjeongeum[1]).

Notwithstanding other critical issues such as fonts (and to a lesser extent indexing [2]), by not relying on spaces between characters or words, the CJK scripts are a challenge to conventional typesetting solutions such as TEX. In fact, the algorithms for line-breaking, which conventionally occurs at spaces, and for word-breaking (hyphenation), become inapplicable.

On a side note, although we consider hereinafter only the CJK writing systems, this discussion can be extended to related scripts such as Tangut and Vietnam's Chữ Nôm.

[1] King Sejong (世宗) introduced hangul in the *Hunminjeongeum* (訓民正音) manuscript (1443–1446).

In this paper, we provide a glance at CJK support with XƎTEX and LuaTEX by giving a minimalistic implementation for these East Asian scripts. This work is both a proof of concept and a pedagogical discussion on how to achieve CJK support as simply as possible with the aforementioned typesetting solutions. Both XƎTEX and LuaTEX support Unicode, which enables us to focus on typesetting issues, leaving encoding and font considerations aside.

The rest of this paper is organised as follows. Technical discussion of the proposed implementation is conducted in Section 2. The state of the art and paper contribution are summarised in Section 3. The paper is concluded in Section 4.

2 A minimalistic implementation

We describe here the proposed minimalistic implementation of CJK support with XƎTEX and LuaTEX step by step in a pedagogical manner:

- paragraph management (Step 1) is addressed in Section 2.1,
- Latin text mingling (Step 2) in Section 2.2,
- Latin text paragraphs (Step 3) in Section 2.3,
- Korean text paragraphs (Step 4) in Section 2.4,
- sophisticated line-breaking (Step 5) in Section 2.5.

"Latin text" here designates text written with the Latin alphabet, or similar; for instance English and French text.

A handful of TEX commands appear hereinafter without being detailed; see [5] for those that are not self-explanatory. The document preamble specifies nothing in particular. The `fontspec` package [12] is loaded for ease of font manipulation, and, as detailed in the rest of this section, since it is considered without loss of generality that the document consists of Chinese or Japanese paragraphs by default, the main font of the document is set accordingly (e.g., `\setmainfont{Noto Serif CJK JP}` [4]).

2.1 Paragraph management

A conventional approach to break long character sequences (i.e., Chinese or Japanese characters in our case) is to insert between each two glyphs a small amount of horizontal space so that TEX can split the sequence across multiple lines (see for instance [15]). Without such extra space, line breaks can in general still occur thanks to hyphenation, but this is not applicable in the case of CJK. We rely on a "scanner" macro to transform a paragraph by interleaving space between its characters. In practice, according to the TEX terminology, this extra space will be a horizontal skip of 0pt width and ±1pt stretch.

The scanner macro is a recursive process that takes one token (e.g., a character) as single parameter and outputs it with on its right extra horizontal space. The recursion stops when the parameter token is the stop signal (more on this later), in which case the macro outputs `\par`, thus triggering the end of the paragraph. The scanner macro `\cjk@scan` is defined as follows:

```
\def\cjk@scan#1{% #1: single token
 \ifx#1\cjk@stop% stop signal detected
  \par% so, complete the paragraph
 \else
  #1% display the current character
  \hskip 0pt plus 1pt minus 1pt\relax% space
  \expandafter\cjk@scan% recursive call
 \fi
}
```

This scanner is started by the `\cjk@scanstart` macro, whose primary objective is to append the stop signal `\cjk@stop` at the end of the paragraph that is about to be transformed. This initial macro takes one parameter: the paragraph to transform. In a pattern matching fashion, a paragraph is taken as a whole by setting `\par` as delimiter for the parameter of the `\cjk@scanstart` macro. This will require inserting `\par` once the paragraph has been transformed, since the `\par` command that ends the paragraph is treated as a delimiter by the macro and thus skipped. In addition, each paragraph needs to be ended by a blank line (or, equivalently, `\par`) for this pattern matching to work. The scanner starting macro is this:

```
\def\cjk@scanstart#1\par{% #1: paragraph
 \cjk@scan#1\cjk@stop% append \cjk@stop
}
```

In this work, paragraphs are considered to be written in Chinese or Japanese by default. Hence, paragraph typesetting mode selection by means of a command such as `\CHJPtext` is not suitable. We rely on the `\everypar` token parameter to trigger the transformation of each paragraph with the scanner previously described. This is simply done with the following assignment:

```
\everypar={\cjk@scanstart}
```

or, in a safer manner [3]:

```
\everypar=\expandafter{\the\everypar
 \cjk@scanstart}
```

An illustration of the result of this paragraph transformation is given in Figure 1 with two traditional Chinese paragraphs.

人民身體之自由應予保障。除現行犯之逮捕
人民因犯罪嫌疑被逮捕拘禁時，其逮捕拘禁

(a)

人民身體之自由應予保障。除現行犯之逮捕由法律另定外，非經司法或警察機關依法定程序，不得逮捕拘禁。非由法院依法定程序，不得審問處罰。非依法定程序之逮捕、拘禁、審問、處罰，得拒絕之。

人民因犯罪嫌疑被逮捕拘禁時，其逮捕拘禁機關應將逮捕拘禁原因，以書面告知本人及其本人指定之親友，並至遲於二十四小時內移送該管法院審問。本人或他人亦得聲請該管法院，於二十四小時內向逮捕之機關提審。

(b)

Figure 1: Before (a) and after (b) paragraph transformation: line breaking now enabled (traditional Chinese text example).

2.2 Latin text mingling

It is often the case that Latin text such as English words, expressions or sentences is mingled within Chinese or Japanese paragraphs. In the paragraph transformation method described so far, spaces, if any, are "gobbled" and never passed as parameters to the scanner macro `\cjk@scan`. This is not a problem for Chinese and Japanese text since, as explained, they do not rely on spaces. But now that we are considering Latin text mingling in such paragraphs, spaces need to be retained since Latin text, such as English, does rely on spaces to separate words, sentences, etc.

Without going too far into the details, to force TeX to also pass spaces as parameters to the scanner macro, spaces need to be made *active*, in TeX terminology. Hence, it suffices to call the `\obeyspaces` macro, whose purpose is exactly to make the space character active, at the beginning of the document. In addition, the scanner macro is refined to avoid adding extra space when the current character is a space:

```
\def\cjk@scan#1{%
 \ifx#1\cjk@stop
  \par
 \else
  #1%
  \if#1\space% no extra space if #1 is a space
  \else
   \hskip 0pt plus 1pt minus 1pt\relax
  \fi
  \expandafter\cjk@scan
 \fi
}
```

An illustration of the result of this refined paragraph transformation is given in Figure 2.

We conclude this section with the following two remarks. First, it should be noted that Latin text mingled within Chinese or Japanese paragraphs is treated just as Chinese or Japanese text: extra space is inserted between glyphs. Therefore, line- and

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの
子孫のために、諸国民との協和による成果と
、わが国全土にわたつて自由のもたらす恵沢
を確保し、政府の行為によつて再び戦争の惨
禍が起ることのないやうにすることを決意し
、ここに主権が国民に存することを宣言し、
この憲法を確定する。そもそも国政は、国民
の厳粛な信託によるものであつて、その権威
は国民に由来し、その権力は国民の代表者が
これを行使し、その福利は国民がこれを享受
する。これは人類普遍の原理であり、この憲
法は、かかる原理に基くものである。われら
は、これに反する一切の憲法、法令及び詔勅
を排除する。We,theJapanesepeople...

(a)

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの
子孫のために、諸国民との協和による成果と
、わが国全土にわたつて自由のもたらす恵沢
を確保し、政府の行為によつて再び戦争の惨
禍が起ることのないやうにすることを決意し
、ここに主権が国民に存することを宣言し、
この憲法を確定する。そもそも国政は、国民
の厳粛な信託によるものであつて、その権威
は国民に由来し、その権力は国民の代表者が
これを行使し、その福利は国民がこれを享受
する。これは人類普遍の原理であり、この憲
法は、かかる原理に基くものである。われら
は、これに反する一切の憲法、法令及び詔勅
を排除する。We, the Japanese people...

(b)

Figure 2: Before (a) and after (b) making spaces active: Latin text mingling now retains spaces (Japanese text example).

word-breaking for mingled Latin text can occur anywhere, and thus no word-breaking by hyphenation will happen. Second, even though no extra space is added after a space character, extra space is still added before a space character. This issue will be tackled in a subsequent section.

2.3 Latin text paragraphs

Because the `\obeyspaces` macro has been called so as to typeset Chinese and Japanese paragraphs, Latin text paragraphs would be typeset just as those, that is, with extra space added between consecutive glyphs (except after spaces). As a result, as explained above, line- and word-breaking would not be satisfactory.

Hence, we next enable the proper typesetting of Latin text paragraphs, that is, paragraphs that include spaces between words. To this end, we define the `\iflatin` conditional statement that will be used to distinguish Latin text paragraphs from others. The flag command `\latinfalse` is called at the beginning of the document to reflect that Chinese and Japanese paragraphs are the norm. Latin text paragraphs are marked as such by calling the flag command `\latintrue` at the beginning of the paragraph. The scanner starting macro `\cjk@scanstart` is adjusted so as to not start the scanner in case the Latin flag is set.

Since the `\obeyspaces` macro has been previously called, spaces are active characters; this setting needs to be reverted in the case of a Latin text paragraph in order to have proper line- and word-breaking. Hence, the scanner starting macro in addition reverts spaces from the active state back to their default state in the case of a Latin text paragraph. The refined code is given next:

```
\newif\iflatin % flag to detect whether to scan
\latinfalse % flag initially set to false
```

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの
子孫のために、諸国民との協和による成果と
、わが国全土にわたつて自由のもたらす恵沢
を確保し、政府の行為によつて再び戦争の惨
禍が起ることのないやうにすることを決意し
、ここに主権が国民に存することを宣言し、
この憲法を確定する。
We, the Japanese people, acting throug
h our duly elected representatives in the N
ational Diet, determined that we shall secu
re for ourselves and our posterity the fruits
of peaceful cooperation with all nations an
d the blessings of liberty throughout this la
nd, and resolved that never again shall we
be visited with the horrors of war through
the action of government, do proclaim tha
t sovereign power resides with the people
and do firmly establish this Constitution.

(a)

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの
子孫のために、諸国民との協和による成果と
、わが国全土にわたつて自由のもたらす恵沢
を確保し、政府の行為によつて再び戦争の惨
禍が起ることのないやうにすることを決意し
、ここに主権が国民に存することを宣言し、
この憲法を確定する。
We, the Japanese people, acting through our duly elected representatives in the National Diet, determined that we shall secure for ourselves and our posterity the fruits of peaceful cooperation with all nations and the blessings of liberty throughout this land, and resolved that never again shall we be visited with the horrors of war through the action of government, do proclaim that sovereign power resides with the people and do firmly establish this Constitution.

(b)

Figure 3: Before (a) and after (b) Latin mode enabling: Latin text now properly typeset (Japanese and English text example).

```
\def\cjk@scanstart#1\par{%
 \iflatin% if Latin text paragraph, don't scan
  \catcode`\ =10% revert \obeyspaces
  #1\par% display the paragraph normally
  \latinfalse% back to default
 \else
  \cjk@scan#1\cjk@stop
 \fi
}
```

An illustration of the result of this refined paragraph transformation is given in Figure 3.

2.4 Korean text paragraphs

Let us now discuss the case of Korean text paragraph typesetting. As mentioned in the introduction, modern Korean relies on spaces to separate words. Hence, Korean text paragraphs are treated as Latin text paragraphs, concretely marked with the `\latintrue` flag. Yet, because Korean glyphs (i.e., hangul or hanja) are wider than Latin ones, the width of spaces is adjusted. In addition, a font switch is also used to select a Korean font since it is common that Korean glyphs are not included in the default font used for Chinese and Japanese paragraph typesetting.

Such settings need to be applied at the beginning of the paragraph, so we need to embed the paragraph into a group for font selection and the adjusted space setting. Therefore, the paragraph starts with a '`{`' token, and thus it is required to leave vertical mode for proper parsing of the paragraph when it is used as the parameter of our macro `\cjk@scanstart` which starts the scanner. Specifically, the problem with starting the paragraph with a command like `{\malgun` (e.g., a font switch) is that TeX is still in vertical mode when it is pro-

日本国民は、正当に選挙された国会における代表者を通じて行動し、
われらとわれらの子孫のために、諸国民との協和による成果と、わが国全
土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争
の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存
することを宣言し、この憲法を確定する。

대통령은 내우·외환·천재·지변 또는 중대한 재정·경제상의 위기에 있어서
국가의 안전보장 또는 공공의 안녕질서를 유지하기 위하여 긴급한 조치가
필요하고 국회의 집회를 기다릴 여유가 없을 때에 한하여 최소한으로 필요한
재정·경제상의 처분을 하거나 이에 관하여 법률의 효력을 가지는 명령을
발할 수 있다.

(a)

日本国民は、正当に選挙された国会における代表者を通じて行動し、
われらとわれらの子孫のために、諸国民との協和による成果と、わが国全
土にわたつて自由のもたらす恵沢を確保し、政府の行為によつて再び戦争
の惨禍が起ることのないやうにすることを決意し、ここに主権が国民に存
することを宣言し、この憲法を確定する。

대통령은 내우·외환·천재·지변 또는 중대한 재정·경제상의 위기에 있어서
국가의 안전보장 또는 공공의 안녕질서를 유지하기 위하여 긴급한 조치가
필요하고 국회의 집회를 기다릴 여유가 없을 때에 한하여 최소한으로 필요한
재정·경제상의 처분을 하거나 이에 관하여 법률의 효력을 가지는 명령을 발할
수 있다.

(b)

Figure 4: Before (a) and after (b) space width adjustment for Korean text: no more overfull horizontal boxes (Japanese and Korean text example).

cessed. Switching to horizontal mode starts a new paragraph and thus triggers `\everypar`, but then with an unmatched '`}`' remaining (i.e., the one corresponding to, say, the font switch) at the end of the paragraph, and thus the parsing error.

For convenience, these Korean text paragraph settings are gathered in a `\korean{}` macro as defined below.

```
\def\korean#1{%
 \latintrue% activate the Latin mode
 \leavevmode% leave the vertical mode
 {% Adjust the space size:
  \spaceskip=\fontdimen2\font plus
    3\fontdimen3\font minus
    3\fontdimen4\font% ×3 stretch and shrink
  \malgun #1% Korean font switch
 }
}
```

Note that this redefinition of `\spaceskip` for the current paragraph would also be applied to Latin text mingled within a Korean paragraph. Furthermore, this font selection process — without necessarily activating the Latin mode and adjusting the space width — could also be used in the case where distinct fonts for Chinese and Japanese text are required.

An illustration of the result of this paragraph typesetting is given in Figure 4. One should note the overfull horizontal boxes which are shown by the two black boxes in the left-hand example, when the space width adjustment has not been applied yet.

2.5 Sophisticated line-breaking

Just as, say, in French, where line breaks are not allowed before the punctuation marks ':', ';', '!' and so on — even though these need to be preceded by a space and are thus typical usages of non-breaking spaces — CJK typesetting forbids breaking lines before punctuation marks such as commas and periods.

We derive in this section a new scanner macro, `\cjk@scanbis`, to address this remaining problem. The approach is simple: refrain from adding extra space after the current character when the next one

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの
子孫のために、諸国民との協和による成果と
、わが国全土にわたつて自由のもたらす恵沢
を確保し、政府の行為によつて再び戦争の惨
禍が起ることのないやうにすることを決意し
、ここに主権が国民に存することを宣言し、
この憲法を確定する。

(a)

日本国民は、正当に選挙された国会におけ
る代表者を通じて行動し、われらとわれらの子
孫のために、諸国民との協和による成果と、
わが国全土にわたつて自由のもたらす恵沢を
確保し、政府の行為によつて再び戦争の惨禍
が起ることのないやうにすることを決意し、
ここに主権が国民に存することを宣言し、こ
の憲法を確定する。

(b)

Figure 5: Paragraph transformation by the original (a) and the new (b) scanner macro: no more line break before a comma (Japanese text example).

is a punctuation mark. At the same time, this new scanner allows us to solve the aforementioned incongruity of extra space being added before a space character in Latin text paragraphs.

To implement this, the new scanner takes two tokens as parameters instead of one: the first parameter is the currently processed token and the second one is the next token in line. The recursive call is also updated since it is now expecting two tokens as parameters instead of one; here it is:

```
\def\cjk@scanbis#1#2{% two tokens passed
 #1%
 \ifx#2\cjk@stop
  \par
 \else
  \if#2、% no extra space before character `、'
  \else\if#2。% idem before character `。'
  \else\if#2\space% idem before a space
  \else\if#1\space% idem after a space
  \else\hskip 0pt plus 1pt minus 1pt\relax
  \fi\fi\fi\fi
  \expandafter\cjk@scanbis\expandafter#2%
 \fi
}
```

Similar additional conditions for other CJK punctuation marks can easily be appended if needed.

One other change is needed: in the scanner macro `\cjk@scanstart`, the initial expression
`\cjk@scan#1\cjk@stop`
is modified to
`\cjk@scanbis#1\cjk@stop`.

An illustration of the effect of this new scanner is shown in Figure 5.

3 State of the art and contribution

Early solutions for supporting the CJK writing systems within the TeX ecosystem include the `CJK` package [6] and the Japanese TeX system pTeX [8]. Although the former provides some support for Unicode, the latter does not. Notably, pTeX supports vertical typesetting [10], while the `CJK` package only partially supports it. Based on the `CJK` package, the BXcjkjatype package [16] provides some support for Japanese typesetting with pdfLaTeX (UTF-8 files). Regarding Korean, the hlatex package [14] enables the processing by LaTeX of KS X 1001 encoded files, and of UTF-8 files via the obsolete TeX extension Omega [11]. Omega also has some support for multi-directional CJK typesetting.

More recent solutions include the xeCJK package [7], which is dedicated to XeTeX (i.e., no LuaTeX support). This package is very large, consisting of more than 14,000 lines of macro code. As of summer 2019, it is only documented in Chinese. Another extensive package, luatex-ja [13], is available, this time restricted to support for Japanese with LuaTeX. Finally, up(LA)TeX [9], another system dedicated to Japanese, can also be cited; it is based on p(LA)TeX, but unlike its predecessor supports Unicode.

Even if the above are more or less complete solutions to the CJK typesetting issue with TeX, we have presented in this paper a very simple solution, which requires neither a separate TeX system such as pTeX nor advanced TeX capacities such as xtemplate, LaTeX3, etc., unlike, for instance, xeCJK. With only a few lines of macro code, we have described how to add basic yet arguably competent support for CJK to both XeTeX and LuaTeX, without differentiation. The XeTeX, LuaTeX flexibility has been retained: no extra layer has been piled on as, for instance, with xeCJK (e.g., the \setCJKmainfont command). Moreover, the complexity induced by packages such as xeCJK is likely to be a threat to compatibility with other packages, as well as with online compilation systems such as those employed by scientific publishers.

4 Conclusions

It is well known that the Chinese, Japanese and Korean writing systems are challenging for typesetting programs such as TeX that were originally designed for Latin text. Various extensions and packages have been proposed to support CJK in TeX, with uneven success. Such solutions are in most cases, if not all, extensive — not to say invasive — additions to the TeX ecosystem. In this paper, relying on the Unicode-capable XeTeX and LuaTeX systems, we have presented and pedagogically discussed a minimalistic solution to this CJK typesetting issue. With only a few lines of macro code, we have shown that satisfactory CJK support can be achieved: paragraph management, Latin text mingling and sophisticated line-breaking are examples of the typesetting issues addressed.

As for future work, given its still rather frequent usage, right-to-left horizontal typesetting would be a useful addition to this discussion of CJK typesetting. Furthermore, although it is a complex issue for TeX, right-to-left vertical typesetting is another meaningful objective as it is ubiquitous for the CJK writing systems.

Acknowledgments

The author is grateful to Takeyuki Nagao (Chiba University of Commerce, Japan) and Keiichi Kaneko (Tokyo University of Agriculture and Technology, Japan) for their insightful advice. This research project is partly supported by The Telecommunications Advancement Foundation (Tokyo, Japan).

References

[1] A. Bossard. *Chinese Characters, Deciphered.* Kanagawa University Press, Yokohama, Japan, 2018.

[2] A. Bossard and K. Kaneko. Experimenting with makeindex and Unicode, and deriving kameindex. In *Proceedings of the GuIT meeting 2018, ArsTeXnica 26*, pp. 55–61, Rome, Italy, October 2018. https://www.guitex.org/home/images/ArsTeXnica/AT026/kameindex.pdf

[3] S. Checkoway. *The everyhook package*, November 2014. Package documentation. https://ctan.org/pkg/everyhook (last accessed August 2019).

[4] Google. Google Noto fonts, 2017. https://google.com/get/noto (last accessed August 2019).

[5] D. E. Knuth. *The TeXbook.* Addison-Wesley, Boston, MA, USA, 1986.

[6] W. Lemberg. *CJK*, April 2015. Package documentation. https://ctan.org/pkg/cjk (last accessed August 2019).

[7] L. Liu and Q. Lee. *xeCJK 宏包 (in Chinese)*, April 2018. Package documentation. https://ctan.org/pkg/xecjk (last accessed August 2019).

[8] K. Nakano, Japanese TeX Development Community, and TTK. *About pLaTeX 2ε*, September 2018. Package documentation. https://ctan.org/pkg/platex (last accessed August 2019).

[9] K. Nakano, Japanese TeX Development Community, and TTK. *About upLaTeX 2ε*, April 2018. Package documentation. https://ctan.org/pkg/uplatex (last accessed August 2019).

[10] H. Okumura. pTeX and Japanese typesetting. *The Asian Journal of TeX* 2(1):43–51, April 2008. http://ajt.ktug.org/2008/0201okumura.pdf

[11] J. Plaice and Y. Haralambous. The latest developments in Ω. *TUGboat* 17(2):181–183, June 1996. `https://tug.org/TUGboat/tb17-2/tb51plaice.pdf`

[12] W. Robertson. *The fontspec package — Font selection for XƎLaTeX and LuaLaTeX*, July 2018. Package documentation. `https://ctan.org/pkg/fontspec` (last accessed August 2019).

[13] The LuaTeX-ja project team. *The LuaTeX-ja package*, November 2018. Package documentation. `https://ctan.org/pkg/luatexja` (last accessed August 2019).

[14] K. Un. 한글라텍 길잡이 *(in Korean)*, April 2005. Package documentation. `https://ctan.org/pkg/hlatex` (last accessed August 2019).

[15] B. Veytsman. *Splitting Long Sequences of Letters (DNA, RNA, Proteins, etc.)*, August 2006. Package documentation. `https://ctan.org/pkg/seqsplit` (last accessed August 2019).

[16] T. Yato. *BXcjkjatype package*, August 2013. Package documentation. `https://ctan.org/pkg/bxcjkjatype` (last accessed August 2019).

Permissions

The placeholder text used in the various illustrations of this article is in the public domain as detailed below.

Figure 1: the placeholder text is the two first paragraphs of Article 8 of the Chinese constitution (1947), written in traditional Chinese.

Figure 2: the placeholder text is the first paragraph of the Japanese constitution (1946), followed by the first few words of the corresponding official English translation.

Figure 3: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946), followed by the corresponding official English translation.

Figure 4: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946), followed by the first paragraph of Article 76 of the South Korean constitution (1988).

Figure 5: the placeholder text is the first sentence of the first paragraph of the Japanese constitution (1946).

◇ Antoine Bossard
Graduate School of Science
Kanagawa University
2946 Tsuchiya, Hiratsuka
Kanagawa 259-1293
Japan
`abossard (at) kanagawa-u dot ac dot jp`

TUG 2019 abstracts

Amine Anane

Arabic typesetting using a Metafont-based dynamic font

Arabic script is a cursive script where the shape and width of letters are not fixed but vary depending on the context and the justification needs. A typesetter must consider those dynamic properties of letters to achieve high-quality text comparable to Arabic calligraphy.

In this talk I will present a parametric font that has been designed as a first step towards such high-quality typesetting. The font is based on the Metafont language which can generate a glyph with a given width dynamically, respecting the curvilinear nature of Arabic letters. It uses an extended version of OpenType to support the varying width of the glyphs. I will demonstrate a graphical tool which has been developed specifically to facilitate the design of such dynamic fonts. As a case study, I will compare a handwritten Quranic text with one generated with this dynamic font. I will conclude by highlighting future work towards a complete high-quality Arabic typesetting.

Takuto Asakura

A TeX-oriented research topic: Synthetic analysis on mathematical expressions and natural language

Since mathematical expressions play fundamental roles in Science, Technology, Engineering and Mathematics (STEM) documents, it is beneficial to extract meanings from formulae. Such extraction enables us to construct databases of mathematical knowledge, search for formulae, and develop a system that generates executable codes automatically.

TeX is widely used to write STEM documents and provides us with a way to represent *meanings* of elements in formulae in TeX by macros. As a simple example, we can define a macro such as

```
\def\inverse#1{#1^{-1}}
```

and use it as `$\inverse{A}$` in documents to make it clear that the expression means “the inverse of matrix A” rather than “value A to the power of -1”. Using such meaningful representations is useful in practice for maintaining document sources, as well as converting TeX sources to other formal formats such as first-order logic and content markup in MathML. However, this manner is optional and not forced by TeX. As a result, many authors neglect it and write messy formulae in TeX documents (even with wrong markup).

To make it possible to associate elements in formulae and their meanings automatically instead of requiring it of authors, recently I began research on detecting or disambiguating the meaning for each element in formulae by conducting synthetic analyses on mathematical expressions and natural language text. In this presentation, I will show the goal of my research, the approach I'm taking, and the current status of the work.

An extended abstract is available at `wtsnjp.com/talk/cicm2019/dc-abstract.pdf`.

Erik Braun

Current state of CTAN

The "Comprehensive TeX Archive Network" is the authoritative place where TeX-related material is collected.

Developers can upload their packages, and the distributions use it to pick up their packages. The TeX Catalogue's entries can be accessed via the website, and all the data can be accessed from mirror servers all over the world.

The talk will give an overview of the current state of CTAN, recent developments, and most common problems. In further discussion, feedback from users and developers is very welcome.

Jennifer Claudio, Sally Ha

A brief exploration of artistic elements in lettering

This non-technical talk explores the stylistic elements of letter forms as used in arts and culture through an examination of elongations and decorations with a focus on the letter E. Samples discussed are derived from the calligraphy of Don Knuth's *3 : 16*, in samples of street art, and in typographic branding.

David Fuchs

What six orders of magnitude of space-time buys you

TeX and METAFONT were designed to run acceptably fast on computers with less than 1/1000th the memory and 1/1000th the processing power of modern devices. Many of the design trade-offs that were made are no longer required or even appropriate.

Federico Garcia-De Castro

An algorithm for music slurs in METAFONT

This paper describes an algorithm that draws beautiful slurs around given notes (or other points to avoid). I have been working on such an algorithm on and off since around 2004 — when commercial music typesetting software did not provide for automatic, let alone beautiful, slurs. Along the way I tried many kinds of approaches, some of them inspired by METAFONT routines such as **superellipse**, the **flex** macro, and the **transform** infrastructure (which, for example, is what slants the `\textsl` font out of a vertical design). The usual fate of these attempts was one of promise followed by interesting development leading to collapse — there usually were too many independent variables interacting chaotically.

Earlier this year I finally found a robust, elegant algorithm. I will present all of the attempts and describe what makes the final algorithm unique, and compare it to the way commercial software does slurs today. This is a graphic presentation, rather than musical.

Shakthi Kannan

XeTeX Book Template

The XeTeX Book Template is a free software framework for authors to publish multilingual books using XeTeX. You can write the content in GNU Emacs Org-mode files along with TeX, and the build scripts will generate the book in PDF. The Org-mode files are exported to TeX files, and Emacs Lisp post-processing is done prior to PDF generation. Babel support with Org-mode TeX blocks allows one to selectively export content as needed. The framework separates content from presentation.

A style file exists for specifying customized page titles, setting margins, font specification, chapter title and text formatting, page style, spacing etc. The framework has been used to publish books containing Tamil, Sanskrit and English. It is released under the MIT license and available at `gitlab.com/shakthimaan/xetex-book-template`.

In this talk, I will explain the salient features of the XeTeX Book Template, and also share my experience in creating and publishing books using the framework.

Doug McKenna

An interactive iOS math book using a new TeX interpreter library

The current TeX ecosystem is geared towards creating only static PDF or other output files. Using a re-implementation of a TeX language interpreter as a library linked into an iOS client program that simulates a document on a device with a touch screen, the author will demonstrate a new PDF-free ebook, *Hilbert Curves*, that typesets itself each time the application launches. The library maintains all TeX data structures for all pages in memory after the typesetting job is done, exporting pages as needed while the user reads the book and interacts with its dynamic illustrations. This design also allows text-searching the document's TeX data structures while the ebook is "running".

Frank Mittelbach

Taming UTF-8 in pdfTeX

To understand the concepts in `pdflatex` for processing UTF-8 encoded files it is helpful to understand the models used by the TeX engine and earlier models used by LaTeX on top of TeX. This talk gives a short historical review of that area and explains

— how it is possible in a TeX system that only understands 8-bit input to nevertheless interpret and process UTF-8 files successfully;
— what the obstacles are that can be and have been overcome;
— what restrictions remain if one doesn't switch to a Unicode-aware engine such as LuaTeX or XeTeX.

The talk will finish with an overview about the improvements with respect to UTF-8 that will be activated in LaTeX within 2019 and how they can already be tested right now.

Ross Moore

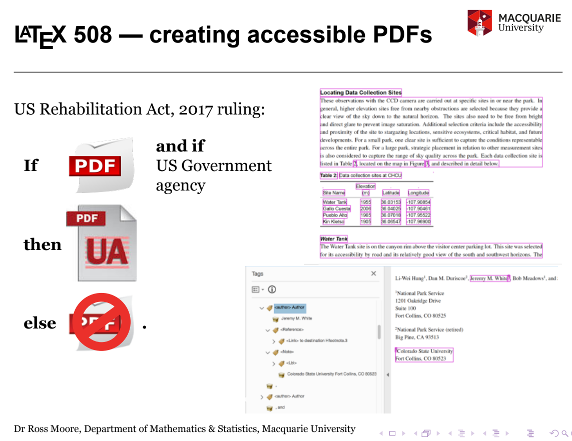
LaTeX 508 — creating accessible PDFs

Authoring documents that are accessible to people with disabilities is not only the morally correct thing to be doing, but is now required by law, at least for U.S. Government offices and agencies, through the revised Section 508 of the U.S. Disabilities Act (2017). It is likely to eventually become so also for any affiliated institutions, such as universities, colleges and many schools.

For mathematics and related scientific fields, it thus becomes imperative that we be able to produce documents using LaTeX that conform to the accessible standard ANSI/AIIM/ISO 14289-1:2016 (PDF/UA-1). This is far more rigorous than standard PDF, in terms of capturing document structure, as well as all content associated with each particular structural element.

In this talk we show an example of a research report produced as PDF/UA for the U.S. National Parks Service. We illustrate several of the difficulties involved with creating such documents. This is due partly to the special handling required to encode the structure of the technical information such as appears on the title page, and inside-cover pages, as well as tabular material and images throughout the body of the document. But there are also difficulties that are due to the nature of TeX itself, and the intricacy of LaTeX's internal programming.

Videos of this talk and another talk on accessibility, by Chris Rowley, are available at `web.science.mq.edu.au/~ross/TaggedPDF/TUG2019-movies`. The basic discussion slide follows:



Shreevatsa R

What I learned from trying to read the TeX program

As we know, TeX is written in a system called `WEB` that exemplifies the idea of literate programming (or programs as literature), and has been published as a book. Indeed, many good and experienced programmers have read the program with interest. But what if the reader is neither good nor experienced? Here we discuss some (more or less superficial) obstacles that stymie the novice modern programmer trying to make sense of the TeX program, and how they can be overcome. Further information is at `http://shreevatsa.net/tex/program`.

Yusuke Terada

Construction of a digital exam grading system using TeX

At our school in Japan, large-scale paper exams are held on a regular basis. The number of examinees is enormous, and the grading must be finished within a short period of time. Improving efficiency was strongly needed. So I developed a digital exam grading system using TeX. TeX and related software play a core role in the system, co-operating with iPad and Apple Pencil.

In this presentation, I would like to present how TeX can be effectively applied to constructing the digital exam grading system. I will also mention the unexpected difficulties that I faced in the actual large-scale operations and the way I have overcome them.

MAPS 49 (2019)

MAPS is the publication of NTG, the Dutch language TEX user group (`http://www.ntg.nl`).

Ferdy Hanssen, Van de penningmeester [From the Treasurer]; pp. 1–2

Frans Goddijn, Verslag 57ste NTG bijeenkomst [Report of the 57th NTG Meeting]; pp. 3–4

Tim van de Kamp, Impressie hackerskamp SHA2017 [Hackerskamp SHA2017: An impression]; p. 5

Last year there was a large hacker camp on the Flevopolder. Because TEXnicians are also allowed to call themselves hackers, there was even a real LATEX village present at the camp this year. This is a short report of the camp.

Hans van der Meer, Take Notes — Take Two; pp. 6–12

Second, revised and extended, version of a ConTEXt module for processing of notes. Notes are classified according to category/subcategory and can contain information about subject, author, date, source, etc. The typesetting of the notes can be filtered according to several criteria. Many aspects of the formatting are easily configurable.

Hans van der Meer, Bits and pieces from ConTEXt mailing list; pp. 13–26

My Takenotes module for processing notes is used to present a selection of notes collected mainly from the ConTEXt users group on the Internet.

Hans Hagen, LuaTEX 1.10, a stable release (in Dutch); pp. 27–28

A brief history of the LuaTEX project, the relationship with ConTEXt, and the new stability of LuaTEX, while engine experiments will continue with a different program.

Hans Hagen, Basic image formats; pp. 29–30

Handling of images as rule nodes in LuaTEX, and a consideration of each of the basic types JPG, PDF, PNG.

Hans Hagen, Is TEX really slow?; pp. 31–34

Sometimes you read complaints about the performance of TEX, for instance that a LuaTEX job runs slower than a pdfTEX job. But what is actually a run? In the next few pages I will try to explain what happens when you process some text and why even a simple TEX job takes about half a second to process on my laptop.

Dennis van Dok, Dagboek van een Informaticus [An Informatician's diary]; pp. 35–36

A recounting of the author's personal history with computing and TEX.

Ernst van der Storm, Belangrijke onderdelen voor een programmaboekje [Important parts for a program booklet]; pp. 37–38

For many years I have been making program booklets for the Nieuwegeins Kamerkoor, and always did so with LATEX or LuaLATEX. This article describes some macros that I used to make the booklet. Aligning lyrics on the page — usually A5 — and the translation thereof is usually manual work. The `verse` package turned out to be unsuitable; this article includes an alternative.

Taco Hoekwater, MuPDF tools; pp. 39–40

The application MuPDF (`http://mupdf.com`) is a very fast, portable, open source PDF previewer and development toolkit actively supported by Artifex, the creators of Ghostscript (`http://artifex.com`). But MuPDF is not *just* a very fast, portable, open source PDF previewer and toolkit. It also comes with a handy collection of command-line tools that are easily overlooked. The command-line tools allow you to annotate, edit, and convert documents to other formats such as HTML, SVG, PDF, and PNG. You can also write scripts to manipulate documents using JavaScript. This small paper gives a quick overview of the possibilities.

Siep Kroonenberg, Een kleine wegwijzer naar TEX documentatie [Finding TEX documentation]; pp. 41–42

Finding the information you need can be difficult, even for TEX and LATEX users. But I hope to show here that you usually don't have to search for long. TEX Live and MiKTEX install almost complete documentation. There are also very complete and searchable overviews online.

Ernst van der Storm, Veel pagina's scannen, één pdf [Scan many pages, produce one pdf]; pp. 43–48

For a choir or an orchestra it is sometimes necessary to copy parts from a music book resulting in a number of scanned images — usually JPEG or PDF. Below I describe a method using a few LATEX macros to make the margins of all pages straight and symmetrical, display the scans on the entire page and make the result available as a single PDF for printing. Correcting the trapezoidal shape of a scan, however, needs more specific software such as DigiKam. Using an editor with column editing options can be useful.

[Received from Wybo Dekker.]

ConTEXt Group Journal 2018

The ConTEXt Group publishes proceedings of the annual ConTEXt meetings.
`http://articles.contextgarden.net`.

[Received from Taco Hoekwater.]

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 `tug.org/consultants.html`. If you'd like to be listed, please see there.

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
+1 213-617-8483
Email: `typesetting (at) dangerouscurve.org`
We are your macro specialists for TEX or LATEX fine typography specs beyond those of the average LATEX macro package. 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.

Dominici, Massimiliano
Email: `info (at) typotexnica.it`
Web: `http://www.typotexnica.it`
Our skills: layout of books, journals, articles; creation of LATEX classes and packages; graphic design; conversion between different formats of documents.

We offer our services (related to publishing in Mathematics, Physics and Humanities) for documents in Italian, English, or French. Let us know the work plan and details; we will find a customized solution. Please check our website and/or send us email for further details.

Latchman, David
2005 Eye St. Suite #6
Bakersfield, CA 93301
+1 518-951-8786
Email: `david.latchman (at) texnical-designs.com`
Web: `http://www.texnical-designs.com`
LATEX consultant specializing in the typesetting of books, manuscripts, articles, Word document conversions as well as creating the customized LATEX packages and classes to meet your needs. Contact us to discuss your project or visit the website for further details.

Sofka, Michael
8 Providence St.
Albany, NY 12203
+1 518 331-3457
Email: `michael.sofka (at) gmail.com`
Personalized, professional TEX and LATEX consulting and programming services.

I offer 30 years of experience in programming, macro writing, and typesetting books, articles, newsletters, and theses in TEX and LATEX: Automated document conversion; Programming in Perl, C, C++ and other languages; Writing and customizing macro packages in TEX or LATEX, `knitr`.

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.

TEXtnik
Spain
Email: `textnik.typesetting (at) gmail.com`
Do you need personalised LATEX class or package creation? Maybe help to finalise your current typesetting project? Any problems compiling your current files or converting from other formats to LATEX? We offer +15 years of experience as advanced LATEX user and programmer. Our experience with other programming languages (scripting, Python and others) allows building systems for automatic typesetting, integration with databases, … We can manage scientific projects (Physics, Mathematics, …) in languages such as Spanish, English, German and Basque.

Veytsman, Boris
132 Warbler Ln.
Brisbane, CA 94005
+1 703 915-2406
Email: `borisv (at) lk.net`
Web: `http://www.borisv.lk.net`

TEX and LATEX consulting, training, typesetting and seminars. Integration with databases, automated document preparation, custom LATEX packages, conversions (Word, OpenOffice etc.) and much more.

I have about two decades of experience in TEX and three decades of experience in teaching & training. I have authored more than forty packages on CTAN as well as Perl packages on CPAN and R packages on CRAN, published papers in TEX-related journals, and conducted several workshops on TEX and related subjects. Among my customers have been Google, US Treasury, FAO UN, Israel Journal of Mathematics, Annals of Mathematics, Res Philosophica, Philosophers' Imprint, No Starch Press, US Army Corps of Engineers, ACM, and many others.

We recently expanded our staff and operations to provide copy-editing, cleaning and troubleshooting of TEX manuscripts as well as typesetting of books, papers & journals, including multilingual copy with non-Latin scripts, and more.

Warde, Jake
Forest Knolls, CA 94933
650-468-1393
Email: `jwarde (at) wardepub.com`
Web: `http://www.myprojectnotebook.com`

I have been in academic publishing for 30+ years. I was a Linguistics major at Stanford in the mid-1970s, then started a publishing career. I knew about TEX from Computer Science editors at Addison-Wesley who were using it to publish products. Beautiful, I loved the look. Not until I had immersed myself in the production side of academic publishing did I understand the contribution TEX brings to the reader experience.

Long story short, I started using TEX for exploratory projects (see the website referenced) and want to contribute to the community. Having spent a career evaluating manuscripts from many perspectives, I am here to help anyone who seeks feedback on their package documentation. It's a start while I expand my TEX skills.

TUG Institutional Members

TUG institutional members receive a discount on multiple memberships, site-wide electronic access, and other benefits:
`tug.org/instmem.html`
Thanks to all for their support!

American Mathematical Society, *Providence*, *Rhode Island*

Association for Computing Machinery, *New York*, *New York*

Aware Software, *Newark*, *Delaware*

Center for Computing Sciences, *Bowie*, *Maryland*

CSTUG, *Praha*, *Czech Republic*

Duke University Press, *Durham*, *North Carolina*

Harris Space and Intelligence Systems, *Melbourne*, *Florida*

Institute for Defense Analyses, Center for Communications Research, *Princeton*, *New Jersey*

Maluhy & Co., *São Paulo*, *Brazil*

Marquette University, *Milwaukee*, *Wisconsin*

Masaryk University, Faculty of Informatics, *Brno*, *Czech Republic*

Nagwa Limited, *Windsor*, *UK*

New York University, Academic Computing Facility, *New York*, *New York*

Overleaf, *London*, *UK*

StackExchange, *New York City*, *New York*

Stockholm University, Department of Mathematics, *Stockholm*, *Sweden*

TEXFolio, *Trivandrum*, *India*

TNQ, *Chennai*, *India*

Université Laval, *Ste-Foy*, *Québec*, *Canada*

University of Ontario, Institute of Technology, *Oshawa*, *Ontario*, *Canada*

University of Oslo, Institute of Informatics, *Blindern*, *Oslo*, *Norway*

VTEX UAB, *Vilnius*, *Lithuania*

Calendar

2019

Sep 4 – 7	Association Typographique Internationale (ATypI) annual conference, Tokyo, Japan. atypi2019.dryfta.com
Sep 15 – 20	XML Summer School, St Edmund Hall, Oxford University, Oxford, UK. xmlsummerschool.com
Sep 16 – 21	13th International ConTEXt Meeting, “Dirty tricks & dangerous bends”, Bassenge, Belgium. meeting.contextgarden.net/2019
Sep 23 – 26	19th ACM Symposium on Document Engineering, Berlin, Germany. www.documentengineering.org/doceng2019
Oct 3 – 6	Ladies of Letterpress + STL Print Week #4, St. Louis, Missouri. ladiesofletterpress.com/conference
Oct 12	TEXConf 2019, Shibuya, Tokyo, Japan. texconf2019.peatix.com
Oct 19	DANTE 2019 Herbsttagung and 61st meeting, Kirchheim unter Teck, Germany. www.dante.de/veranstaltungen/herbst2019
Oct 26	GuIT Meeting 2019, XVI Annual Conference, Turin, Italy. www.guitex.org/home/en/meeting
Oct 25	Award Ceremony: The Updike Prize for Student Type Design, Providence Public Library, Providence, Rhode Island. www.provlib.org/updikeprize

2020

Feb 28 – Mar 1	Typography Day 2020, “Typographic Dialogues: Local-Global”. Beirut, Lebanon. www.typoday.in
Mar 25 – 27	DANTE 2020 Frühjahrstagung and 62nd meeting, Lübeck, Germany. www.dante.de/veranstaltungen
Apr 24 – 25	Before & Beyond Typography: Text in Global & Multimodal Perspective, Stanford University, Stanford, California. www.eventbrite.com/e/before-beyond-typography-text-in-global-multimodal-perspective-tickets-69068930029
May	BachoTEX 2020, 28th BachoTEX Conference, Bachotek, Poland. www.gust.org.pl/bachotex
Jun 4 – 6	Markup UK 2020. A conference about XML and other markup technologies, King’s College, London. markupuk.org
Jul 1 – 3	Eighteenth International Conference on New Directions in the Humanities, “Transcultural Humanities in a Global World”, Ca’ Foscari University of Venice, Venice. Italy. thehumanities.com/2020-conference
Jul 19 – 23	SIGGRAPH 2020, “Think beyond”, Washington, DC. s2020.siggraph.org
Jul 22 – 24	Digital Humanities 2020, Alliance of Digital Humanities Organizations, Carleton University and the University of Ottawa, Ottawa, Canada. adho.org/conference

TUG 2020 Rochester Institute of Technology, Rochester, New York

Aug	The 41st annual meeting of the TEX Users Group. tug.org/tug2020

Status as of 15 September 2019

For additional information on TUG-sponsored events listed here, contact the TUG office (+1 503 223-9994, fax: +1 815 301-3568, email: office@tug.org). For events sponsored by other organizations, please use the contact address provided.

User group meeting announcements are posted at tug.org/meetings.html. Interested users can subscribe and/or post to the related mailing list, and are encouraged to do so.

Other calendars of typographic interest are linked from tug.org/calendar.html.

Introductory

Intermediate

Intermediate Plus

Advanced

Reports and notices