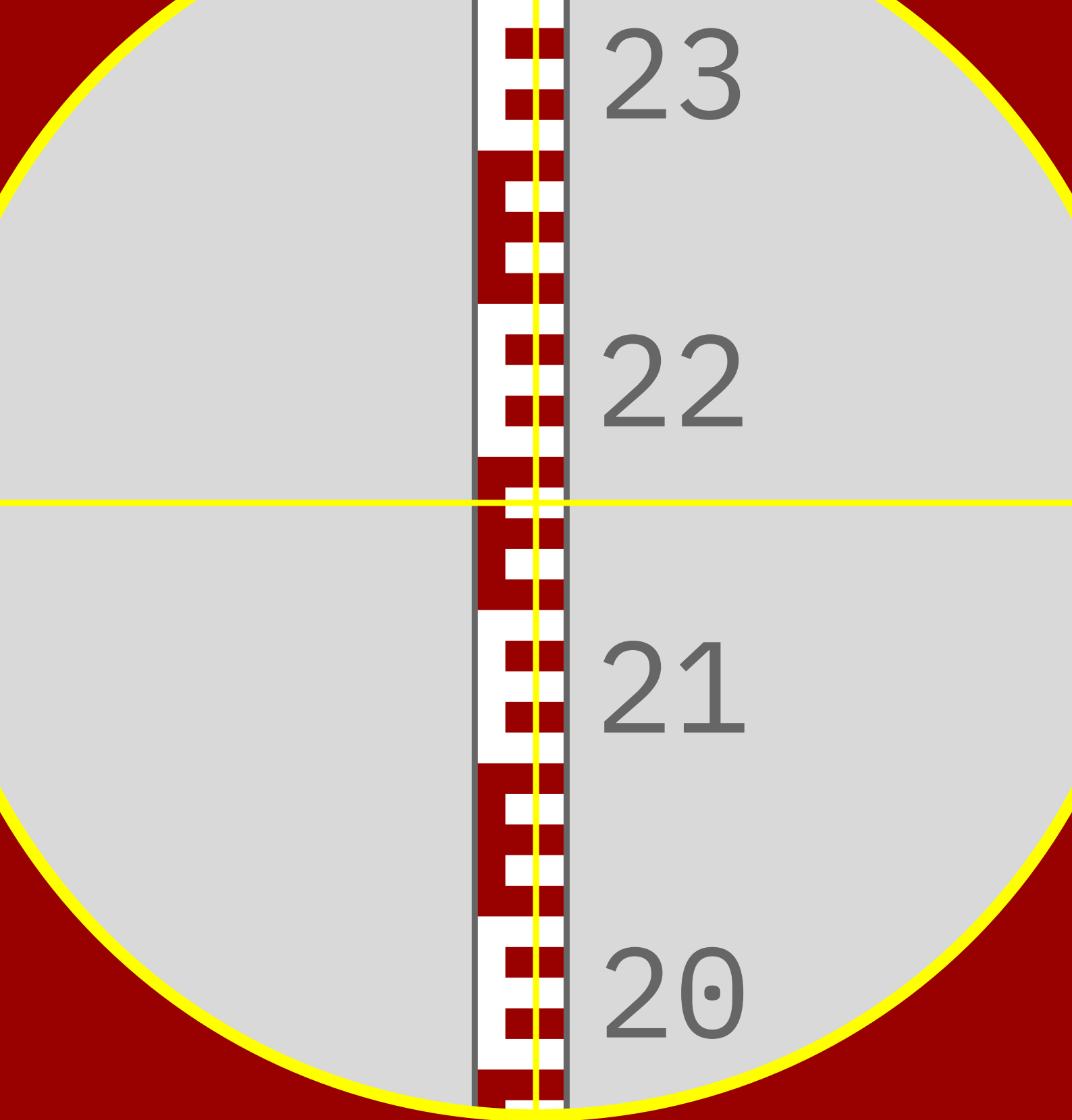


on target

luametatex & context lmtx

Table of contents

1 Introduction 4

2 Eventually 1.0 6

3 A new unit: dk 12

4 Anchoring 14

5 A different approach to math spacing 18

6 The binary 50

7 To the point 52

8 Not all makes sense 64

9 But this does 70

10 The curious case of \over 72

11 Getting rid of jit 74

12 Issues in math fonts 76

13 Gaining performance 90

14 LMTX on a phone 94

15 Running green 96

16 Supporting math in the JMN collection 100

17 Profiles 104

18 Pushing the envelope 112

19 Dealing with math fonts 130

20 The three math sizes 146

21 Constants 152

22 Active characters 156

23 Accuracy 160

24 Characters in math 164

25 Standardizing math fonts 168

26 Somewhat radical 178

27 Between bars 182

28 Getting rid of math rules 186

29 Unusual bitmaps 192

30 Crossing rivers 214

31 Including PDF files 218

32 Signing documents 224

33 Bitmap fonts 232

34 Meaningful math 246

35 Getting nostalgic 260

36 PDF 2.0 264

1 Introduction

This is the seventh wrapup of the LuaTEX and LuaMetaTEX development cycle. It is dedicated to all those users who kept up with developments and are always willing to test the new features. Without them a project like this would not be possible.

At the time this introduction is written the LuaMetaTEX code base is rather stable and quite a bit of the MkIV code base has been adapted to new situation. But, as usual, there are always new possibilities to explore, so I expect that this document will grow over time as did the others. I'm not going to repeat all that has been done because that's what the previous episodes are about.

As the title suggest, we're still on target. When the LuaMetaTEX project started there actually was no deadline formulated so in fact we're always on target. The core components TEX, MetaPost, and Lua are all long term efforts so we're in no hurry at all. However, this is the year that a fast pace will become a slow pace with respect to the LuaMetaTEX code base. There are still some things on the agenda but in principle the goals are reached. One problem in today's code development is that useability and quality seems to relate to the amount of changes in code. No update can mean old, unusable and uninteresting. It's probably why some sources get this silly yearly copyright year update. However, the update cycle of good old TEX has an decade interval by now while it is still a pretty useable program. It would be nice to end up in such a long term cycle with LuaMetaTEX: bug fixes only.

Although ConTEXt has always adapted early to new developments (color, graphics, pdf, MetaPost, e-TEX, pdfTEX, LuaTEX, utf, fonts) the effects on the ConTEXt code base are mostly hidden for users. There have been some changes between MkII and MkIV, simply because there has been a shift from specific eight bit encodings to utf and Type1 to OpenType fonts. Both had an impact on important subsystems: input encodings, font definitions and features, language and script support. On other subsystems the impact was hardly noticeable, like for instance backend related features (these have always been kind of abstract). That doesn't mean that these haven't changed deep down, they definitely have. Some mechanisms became better in MkIV, simply because less hackery was needed. My experience is that when users see that it gets better or easier, they are also willing to adapt the few lines in their document source that benefit from it. Of course the impact on the MetaPost integration in ConTEXt had a real large impact, especially in terms of performance.

The upgrade to LMTX, the version of ConTEXt for LuaMetaTEX, is even less visible although already some new mechanisms showed up. This time a couple of engine specific features have been improved and made more flexible. In fact, the whole code base of the engine has been overhauled. This happened stepwise because we had to make sure all things kept working. As a first step code was made independent of the compilation infrastructure and the dependencies, other than a very few small ones, have been removed. The result is a rather lean and mean setup, even when we consider what has been added at the primitive level and traditional subsystems. A benefit is that in the meantime the LuaMetaTEX LMTX combination outperforms LuaTEX with MkIV, something that was not ensured when the built-in pdf backend was removed and delegated to Lua. By binding development closely to ConTEXt we also hope that the code base stays clean of arbitrary extensions.

Because in the end, TEX is also a programming language, there have been extensions that make programming easier. There is already a stable middle layer of auxiliary macros in ConTEXt that help the user who likes to program but doesn't like real low level primitives and dirty tricks, but by extending the primitive repertoire a bit users can now stay closer to the original TEX concepts. Adding more and more layers of indirectness makes no sense if we can improve the bottom programming layer. It also makes coding a bit more natural (the TEX look), apart from offering performance benefits. This is where you can see differences between the MkIV and LMTX code base which for that reason is now nearly split completely. The MetaPost

subsystem has been extended with proper scanners so that we can enhance the interfaces in a natural way and as a result we also have an upgraded code base there. We also moved to Lua5.4 and will keep up as long as compatibility is no issue. Some Lua code is likely to remain common between MkIV and LMTX, for instance font handling and helpers but we'll see where that ends.

The LuaMetaTEX engine provides control over most internals and there are all kind of new interesting features. Decades of ConTEXt development are behind that. Also, in the days that there were discussions about extending TEX, ConTEXt was not that much of influence and on the road to and from user group meetings, Taco and I often discussed what we'd like to see added (and some was actually implemented in `eetex` but that only lived on our machines. One can consider LuaTEX to be a follow up on that, and LuaMetaTEX in turn follows up on that project, which we both liked doing a lot. In some way LuaTEX lowered the boundary for implementing some of the more intrusive extensions in LuaMetaTEX and the follow up on mplib. And once you start along that road small steps become large steps and one can as well be try to be as complete as possible. We've come a long way but eventually arrived at the destination. Personally I think we got there by not being in a hurry. Even targets that are (nearly) reached can eventually move,

I want to use this opportunity to thank Karl Berry. When making some of the stories presented here ready for TugBoat, he not only turns bad English into good, but also checks and feeds back on what we write. We really appreciate that and it also motivates us to keep wrapping up.

Hans Hagen
Hasselt NL
August 2021^{++}

2 Eventually 1.0

2.1 Reflection

This is just a short reflection on how we came to version 1.0 of LuaMetaTEX. Much has already been said in articles and history documents. There is nothing in here that is new but I just occasionally like to wrap up the current state. At the time of writing, which happens to be the ConTEXt 2021 meeting, we're somewhere between 0.9 and 1.0 and as usual it reflects a current state of mind.

2.2 Introduction

The development on LuaMetaTEX took a bit more time than I had in planned when I started with it. I presume that it also relates to the way the TEX program is looked at: a finished program that converges to a bugless state. But, with version 1.0 near by it makes sense to reflect on the process. Before I go into details I want to remark that when I wrote ConTEXt I looked at this program from the macro end. I had no real reason to look into the code, and figuring out what happens in a black box is a challenge (and kind of game) in itself. At the time I started using TEX I had done my share of complex and relatively large scale programming in Pascal and Modula so it's not that I was afraid of languages. It was before the Internet took off and not being in academia and connected one had to figure things out anyway. I did have Don's 5 volume TEX series but stuck to the TEX book. Being on msdos I couldn't compile the program anyway, definitely not without the source at hand. I did read the first chapters of the MetaFont book, but apart from being intrigued by it, it was not before I ran into MetaPost that knowing that language took off. Of course I had browsed TEX the program but not in a systematic way.

I was involved with pdfTEX development but stayed at my end of the line: needs, applications, testing and suggestions. With LuaTEX that line got crossed, triggered by the Lua interfaces, but while I focussed on the TEX end, Taco did the C, and we had pleasant and intense daily discussion on how to move forward. I could not get away any longer with the abstraction but had to deal with nodes and such, which was okay as we were hit the boundaries of convenience programming solutions in ConTEXt.

When we started our LuaTEX journey the TEX follow-up most widely used, pdfTEX, did have some e-TEX extensions but in retrospect only a few of those were of relevance to us, like the concept of `\protected` macros[1] and the larger set of registers. And the e-TEX project, in spite of occasional discussions, never became a continuous effort. The nts project that was related to e-TEX and had as objective an extensible successor produced a Java implementation but that one was never useful (as a starter, its performance was such that it could not be used) and I didn't really look forward to spending time on Java anyway. Taco and I played with an extended e-TEX but lack of time made that one end up in the archive.

There were some programmatic additions to pdfTEX but it's main attributes were protrusion, expansion and a pdf backend (Hàn Thế Thành's thesis subject). Features like position tracking were handy but basically just a built-in variant of a concept we already had come up with at the dvi level (using a postprocessing script that later became `dvipos`). There was Omega with a directional model but this engine was always more of an academic project, not a production system.[2] It was XHTEX that moved the TEX world into the Unicode domain and opened the engine up to new font technologies. Although utf8 was already doable in earlier engines (which is why ConTEXt used it already for some internals), native support was way more convenient.

[1] In ConTEXt we always had a protection mechanism and from the LuaTEX source I learned that the macro bases solution was basically the same as the one used in the engine.

[2] Aleph was more reliable but never took off, if only because pdfTEX had a backend.

It was clear that if we wanted to move on we had to make more fundamental steps, but in such a way that it still fit in with what people expect from TEX. While it started an a playground by embedding the Lua interpreter, it quickly became clear that we could open up the internals in fundamental ways, thereby also getting around the discussion about to what extent TEX could and should be extended: that discussion could be and was postponed by the opening up. Because we already foresaw some of possibilities it was decided to freeze ConTEXt for the older engines. It was around the first ConTEXt meeting that the MkII and MkIV tags showed up, around the same time that LuaTEX became useable. More than a decade later, when LuaTEX basically had become frozen, at another meeting it was decided to move on with LuaMetaTEX: the LuaTEX project was pretty much a ConTEXt projects and that follow up would be even more driven by ConTEXt users and usage. But how does it all feel 15 years later? I'll try to summarize that below. It will also explain why I got more audacious in extending the LuaTEX engine into what is now LuaMetaTEX. This also related to the fact that at some point I realized that progress just demands taking decisions, and it happens that we can make these in the perspective of ConTEXt without side effects for other TEX usage. It is also fun to experiment.

2.3 Extending necessary parts

The pdfTEX program, having a backend built in already supports the usage of wide TrueType but it was XETEX that first provided using them directly in the frontend. But that happened within the concept of traditional TEX, especially when it comes to math. There are some extra primitives to deal with scripts and languages but (and this is personally) I decided that these didn't really fit in the way ConTEXt looks at things so MkII doesn't support anything beyond the fonts. The XETEX program first was available on Apple computers and font support was closely related to its technology as well as technologies that relate to where the program originates. Later other operating systems became supported too.

We decided in LuaTEX to delegate 'everything fonts' to Lua, for a good reason: we didn't want to be platform dependent. And using libraries has the danger of periodical enforced fundamental changes because in these times software politics and fashion have short cycles. The fact that XETEX later changed the font engine proved that this was a good decision. At some point LATEX decided to use a special version of LuaTEX that uses a font library as alternative, which is fine, but that also introduces a dependency (and frequent updating of the binary). The LuaTEX engine has a slim variant of the FontForge library built in for reading various font formats and its backend can embed subsets of OpenType, Type1 and traditional bitmap fonts. At some point ConTEXt switched to its own Lua based font file interpreter and experimented with a Lua based backend that later became exclusive for LuaMetaTEX. It became clear that we could do with less code in the engine and thereby less dependencies.

In this perspective it is also good to notice that the LuaTEX engine has no real concept of Unicode: it just expects utf8 and that's it. All internals provide enough granularity to support Unicode. The rest has to come from the macro package, as we know that each one does it its own way. There are no dependencies on Unicode libraries. You only have to look at what ends up on your system when you install a program that just juggles bytes to notice that by including one library a whole lot gets drawn in, most of which is not relevant to the program and we don't want that. It might start small but who knows where one ends up. If we want users to be able to compile the program, we don't want to end up in dependency hell.

The LuaTEX project was, apart from curiosity and potential usage in ConTEXt, initially also driven by the Oriental TEX project that aimed at high quality bidirectional typesetting. There the focus was on fonts as well as processing paragraphs. That triggered all kinds of opening up of internals and once ConTEXt started swapping (and adding) mechanisms using Lua more came to fruit. In the end it took a decade to reach version 1.0 and we could have stopped there knowing that we're quite prepared for the future.

Although the whole TEX concept didn't change, there were some fundamental changes. From the documentation by Don Knuth it becomes clear that interpreting is closely interwoven with typesetting: the so

called main interpretation loop calls out to font processing, ligature building, hyphenation, kerning, breaking lines, processing pages, etc. In LuaTEX these steps became more independent simply because the processing of fonts (via Lua) came down to feeding a linked list of nodes to a callback function. That list should be hyphenated if needed (a now separated step) and if needed the traditional font processing could be applied (ligature building and kerning). But, although one can say that we already got away from the way TEX works internally, most documentation to the original program still applied, simply because the fundamental approach was the same. We didn't feel too guilty about it and I don't think anyone objected. By the way, the same is true for the math subsystem: we had to adapt it to OpenType parameters and formula construction and although that was inspired by TEX it definitely was different, even to the extend that the math fonts that evolved in the community are now a strange hybrid of old and new.

2.4 Getting around the frozen machinery

So why did the LuaMetaTEX project started at all? There has been plenty written on how LuaTEX evolved and the same is true for LuaMetaTEX so I'm not going to repeat that here. It is enough to know that the demand for a stable and frozen LuaTEX by other users than ConTEXt simply doesn't go well with further experiments and we still had plenty ideas. Because at some point Taco had no time I was already responsible for quite some additions to the LuaTEX program so it was no big deal to switch to a an even more extensive mix of working with "TEX the macro language" and "TEX the program".

The first priorities were with some basic cleanup: remove unused font code, get rid of some ever changing libraries and remove the backend related code. I could do that because I already had a Lua driven backend in MkIV (which was removed later on) and font handling was already all done in Lua. The idea was to go lean and mean, and indeed, even with all kind of extensions, the binary is much smaller than its predecessor, which is nice because it is also a Lua engine. Simplifying the build so that users can easily compile themselves was also of high priority because I considered the rather large and complex setup as a time bomb. And I also had my doubts if we could prevent the LuaTEX engine to evolve over time in a way that made it less useable for ConTEXt.

But, interestingly all this extending and pruning didn't feel like I was violating the concept of a long term stable engine. In fact, original TEX has no backend either, just a simple binary serialization of output (dvi). And by removing some font related frontend code we actually came closer to the original. I suppose that these decisions slowly made me aware of the fact that there was no reason to not consider more drastic extensions. After all, wasn't the e-TEX project also about extending.[3]

When we look at LuaMetaTEX 1.0 we still see the expected machinery there but many subsystems have been extended. Once I made the decision that it's now or never, each subsystem got evaluated against my long term wish list and usage in ConTEXt. Now, let's be clear: I basically can do all I want in LuaTEX but that doesn't mean it's always a pretty solution. And to make the ConTEXt code base better to understand for users, even if it is already rather consistent and set up to be readable, is one of my objectives. I spend a lot of time on readability: I cannot stand a bad looking source and over time the look and feel is also determined by the way the ConTEXt interfaces and related syntax highlighting evolved, especially the TEX, MetaPost, Lua mix. This is why LuaMetaTEX has some extensions to the macro language.

So, while some might argue that "It can already be done." I decided to ignore that argument when the actual solutions came too close to "See how well I can do this using dirty tricks!". If we can do better, without harming the system, let's do it: Lua did it, C did it and even Don Knuth switched from Pascal to C. If we want we can put all the extensions under the "TEX is meant to be extended" umbrella, as long as we call it different,

[3] Although non of the ideas that Taco and I discussed on our numerous trips to meetings all over the world ever made it into that engine.

which is what we do. But I admit that one has to (emotionally) cross a boundary of feeling comfortable with fundamental additions to a program like TEX. But I've been around long enough to not feel guilty about it.

So in the end that means that for instance marks were extended, inserts got more options, glyphs and boxes have way more properties, (the result and handling of) paragraphs can be better controlled, page breaking got hooks (and might be extended), local boxes got redone, adjustments were extended, the math machinery has been completely opened up, hyphenation became more powerful, the font mechanism got more control and new scaling features, alignments got some extensions, we can do more with boxes, etc. But often I still first had to convince myself that it's okay to do so. After all, none of this had happened before and to my knowledge also has not been considered in ways that resulted in an implementation (but I might be wrong here). It helps that I can test out experiments in production versions of LMTX and that users are quite willing to test.

2.5 Extending the macro language

In the previous section some mechanisms were mentioned, but before TEX even ends up there macros and primitives come into play. The LuaTEX engine already has some handy extras, like ways to prepend and append tokens and a limited so called 'local control' mechanism (think of nested main loops). There are some new look head and expansions related primitives and csname related tricks. There are a few more conditionals too. Details can be found in manual and articles.

In LuaMetaTEX some more got added and some of these mechanism could be improved and the reason again is that I aim at readable code. Most programming languages for instance have conditionals with some kind of continuation (like `elseif`) and so I added that to TEX too `\orelse`. Actually, there are even more new conditionals than in LuaTEX. Yes, we don't really need these, especially because in LuaMetaTEX we can now extend the primitive language via Lua, but I wanted to improve readability deep down in ConTEXt. It also reduces the clutter when logging, although logging itself has been quite a bit overhauled. There is less need for intermediate (often not that natural) intermediate layers when we can do it properly in primitive TEX lingua.

More fundamental was extending the way TEX deals with macro arguments. Although the extensions to parsing them are using specifiers that make them upward compatible I admit that even I have to consult a list of possibilities every now and then but in the end they make things better (performance wise with less code). As a side effect the macro machinery could be optimized a bit (expansion as well as the save stacking).

There are a few more ways to store integers and dimensions (these fit in nicely), there are new into grouping, some primitives have more keywords and therefore scanners have been extended, the e-TEX expression handlers have alternative variants.

Although this is a sensitive aspect of TEX when it comes to compatibility, at some point I decided that it made no sense to not expose more details about nodes, input, and nesting states. The grouping and input related stacks had been optimized in the meantime so reporting in that area was already not compatible. Improving logging is an ongoing effort and I don't really loose sleep over it not being compatible, as long as it gets better. There is now also some tracing for marks, inserts, math and alignments.

2.6 Refactoring the code base

This is again an emotionally laden decision: what to touch and keep. For sure we keep the original comments but that doesn't make it literate. We started out with a C base that came from converted Pascal web.

The input machinery is a bit different due to the fact that Lua can (and often has to) kick in. In LuaMetaTEX it's even more different because even more goes via Lua. We cannot even run the engine without a basic set of callbacks assigned: if you don't like that, use LuaTEX. Does this violate the TEX concept? Not really, because system dependencies are explicitly mentioned as such in the source code. We have to adapt to the way an operating system sees files anyway (eight bit, utf8, utf16).

We still have many global variables (a practical Knuth thing I guess) but now they are grouped into structures so that we can more clearly see where they belong. This involved quite a but of shuffling and editing but I got there. In LuaMetaTEX all constants (coded in macros) became enumerations, and all hard coded values too which was quite a bit of work too. Probably no one will notice or realize that, but starting from an existing code base is more work than starting from scratch, which is what I always did so far. When possible we use case statements. Most macros became (inline) functions. Complex functions got better variable names. All functions are in name spaces. This was (and is) a stepwise process that takes lots of time, especially because ConTEXt users expect a reasonable stable system and changes like that are sensitive for errors.

Talking of errors, the error and reporting system has been overhauled, so for instance we have now a dedicated string formatter. This all happened in several steps: normalization, consistency, abstraction, formatters, etc. Keep in mind that we not only have the original messages but also new ones. And we have TEX, Lua and MetaPost communicating with the user. Where in LuaTEX we have to conform more to the traditional engine, because that is what other macro packages rely on, In ConTEXt we have more freedom, so we can make it better and more detailed. Of course it could all be controlled by configurations but at some point I decided to kick out variables doing that because it made no sense to complicate the code base.

Memory management has been overhauled (more dynamic) as has dumping to the (more efficient) format file. With what is mentioned in the previous paragraphs we can safely say that in the meantime back porting to LuaTEX (which I had in mind) makes no sense any longer. There is occasionally some pressure to let LuaTEX do the same as other engines (new common features) and that doesn't always fit into the model. There is no need for LuaMetaTEX to follow up on that because often we already have plenty of possibilities. There is of course still work todo, for instance I still have to make some variable names in functions more verbose but that is not fundamental. I also have to go over the documentation in the code. I might make some interfaces more consistent anyway, so that also would demand adaptations. And of course the documentation in general always lags behind.

So far I only mentioned dealing with TEX, but keep in mind that in LuaMetaTEX we also have an upgraded MetaPost: only a Lua backend (we can produce pdf from that other output), no font code, a couple of extensions, more callbacks, io via Lua. Scanners make extending the language possible and injectors make for efficient piping back to MetaPost. Such extensions are also possible in TEX and the LuaMetaTEX scanning interfaces have been improved and extended too. We have extra callbacks (but some were dropped), more helpers (most noticeable in the node namespace), libraries that improve dealing with binary files, a reworked token library (which in turn lead to a reorganization of command codes in the TEX engine), a few more extensions if Lua file handling and string manipulations. We got decimal math, complex math, new compression libraries, better (Lua) memory management, a few optional library interfaces, etc. Fortunately that all didn't bloat the binary.

So, because in the meantime LuaMetaTEX is quite different from LuaTEX, we can consider the last one to be a prototype for the real deal.

2.7 Simplifying the build

This was one of the first things I did. It was a curious process of removing more and more of the original build (all kind of dependencies) which is not entirely trivial because of the way the LuaTEX build is set up.

I admit that I did try to stay within the regular source build concept but after a while I realized that this made no sense so we (Mojca was involved in that) made the move to cmake. Shortly after that I started using Visual Studio as editing environment (which saves time and is rather convenient) and native compilation under MS Windows became possible without any special measures (in fact, setting up the build for arm processors was more work).

A side effect is that right from the start we could provide binaries for various platforms via the compile farm on the ConTEXt garden maintained by Mojca, who also does daily TEX live builds there. On my machine I use the Windows Linux Subsystem for cross compilation but we can also do native builds. And, with my laptop being a robust 2013 old timer I force myself to make sure that LuaMetaTEX keeps performing well.

2.8 Because it just makes sense

So, in the end LuaMetaTEX is likely the engine most different from the Knuthian original but from the above one can conclude that this was a graduate process where I got more audacious over time. In the end the only thing that matters (and I believe that Don Knuth agrees with this) that you like writing the code, feel confident that the code is all right, explore the possibilities, try to improve the quality and understanding and that successive rewrites can reduce obscurity. And in my opinion we didn't loose the TEX look and feel and still can operate well within the established boundaries of the TEX ecosystem. The fact that most ConTEXt users in the meantime use LuaMetaTEX and the related LMTX variant is an indication that they are okay with it, and that is what matters most.

3 A new unit: dk

At the ConTEXt 2021 meeting I mixed my TEX talks with showing some of the (upcoming) LuaMetaTEX source code. One evening we had a extension party where a new unit was implemented, the dk. This event was triggered by a remark Hraban [Ramm] made on the participants list in advance of the meeting, where he pointed to a Wikipedia article from which we quote:

> "In issue 33, Mad published a partial table of the "Potrzebie System of Weights and Measures", developed by 19-year-old Donald E. Knuth, later a famed computer scientist. According to Knuth, the basis of this new revolutionary system is the potrzebie, which equals the thickness of Mad issue 26, or 2.2633484517438173216473 mm [...]."

So, as the result of that session, the source code now has this comment:

> "We support the Knuthian Potrzebie, cf. en.wikipedia.org/wiki/Potrzebie, as the dk unit. It was added on 2021-09-22 exactly when we crossed the season during an evening session at the 15th ConTEXt meeting in Bassenge (Boirs) Belgium. It took a few iterations to find the best numerator and denominator, but Taco Hoekwater, Harald Koenig and Mikael Sundqvist figured it out in this interactive session. The error messages have been adapted accordingly and the scanner in the Lua tex library also handles it. One dk is 6.43985pt. There is no need to make MetaPost aware of this unit because there it is just a numeric multiplier in a macro package."

When compared to the already present units the dk nicely fills a gap:

unit	points	scaled	visual
sp	0.00002	1	
pt	1.0	65536	
bp	1.00374	65781	
dd	1.07	70124	
mm	2.84526	186467	
dk	6.43985	422042	
pc	12.0	786432	
cc	12.8401	841489	
cm	28.45274	1864679	
in	72.26999	4736286	

Deep down, the unit scanner uses a numerator and denominator in order to map the given value onto the internally used scaled points, so the relevant snippet of C is:

```
*num   = 49838; // 152940;
*denom =  7739; //  23749;
return normal_unit_scanned;
```

The impact on performance of scanning an additional unit can be neglected because the scanning code is a bit different from the code in LuaTEX and (probably the) other engines anyway.

Under consideration are a few extra units in the relative_unit_scanned category that we see in css: vw (relative to the \hsize), vh (relative to the \vsize), maybe a percentage (but of what) and ch (width of the current zero digit character). As usual with TEXies, once it's there it will be (ab)used.

4 Anchoring

4.1 Introduction

It is valid to question what functionality should be in the engine and what can best be implemented using callbacks and postprocessing of lists (and boxes) relying for instance on attributes as signals. In LuaTEX we are rather strict in this and assume that the second method is used. In LuaMetaTEX we still promote this but at the same time some (lightweight) functionality has been added to the engine that helps implementing some features more efficiently. Reasons are that it can be handy to carry (fundamental) properties around that are bound to nodes and that we can set them using primitives, especially for glyphs and boxes. That way they become part of the formal functionality that one can argue should be present in a modern engine. Examples for glyph nodes are scales, offsets and hyphenation, detailed ligature and kerning control. For box nodes we have for instance offsets and orientation. Most of these are always taken into account by core mechanisms like breaking paragraphs into lines, where dimensions matter in which case it really makes sense for them to be part of the engine design.

Some properties are just passed on to for instance a font handler or the backend but still they belong to the core functionality. An example of the later is a (new) simple mechanism for anchoring boxes. This is not really a fundamental feature, because one can just move content around using a combination of kerning and boxing, either or not with offsets. But because we already have features like offsets to boxes it was not that much work to add anchoring as a more fundamental property. The frontend is agnostic to this feature because dimensions are kind of virtual here: the backend however carries the real burden. Because backends are written in Lua it might have a performance hit simply because at least we need to check if this feature is used. Normally that can compensated when this feature *is* used because less work and shuffling around happens in the frontend. And when this feature is no longer experimental (and stays) we can gain some back by using it in existing scenarios. It sounds worse than it is because for orientations we already have to do some usage checking and we can share that check; in most situations nothing needs to be done anyway.

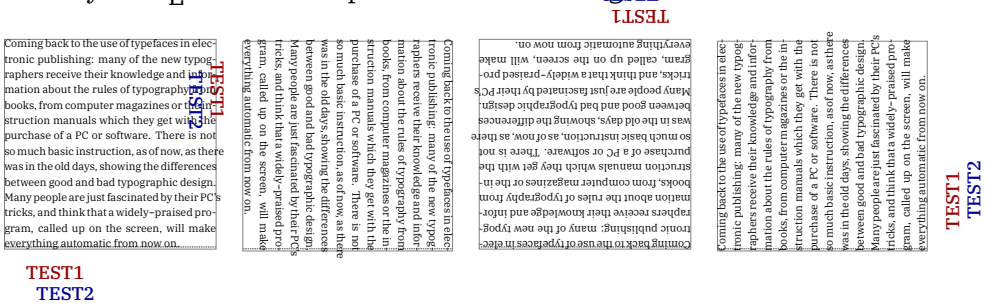
4.2 The low level approach

When we anchor, a box can be a source and/or a target. Both are represented by a number and can be assigned via a keyword. These numbers can be picked up by the backend. Here is an example:

```
\def\TestMe#1{%
    \setbox \scratchbox \ruledvbox
        source 123
        orientation #1
        \bgroup
            \hsize7cm
            \samplefile{zapf}
            \hbox to 0pt
                source 124 target 123
                xoffset 20pt yoffset -30pt
                {\darkred \bfc TEST1}%
            \hbox to 0pt
                source 125 target 124
                xoffset 10pt yoffset -20pt
                {\darkblue \bfc TEST2}%
```

```
        \egroup
    \box \scratchbox
}
```

This example also uses a few offsets. The 'origin' is at the left edge of the baseline. Now, we could have passed the source and target as attribute and intercepting an attribute in the backend can work pretty well. However, the code that deals with the final result of the typesetting and thereby flushes it to for instance a pdf file is, at least that is the setup we use in ConTEXt, attribute agnostic. Mixing in attributes at that stage, except for user nodes and whatsits that are effectively plugins, is counter intuitive and all is already pretty complex so a clear separation of functionality makes a lot of sense. Of course the ConTEXt approach is not the only one when it comes to generic engine functionality. Not that many fundamental (conceptual) extensions showed up over the last few decades so no one will bother if in LuaMetaTEX we have new stuff that is only used by ConTEXt. The example code shown here gives:




```
orientation 0  orientation 1  orientation 2  orientation 3
```

In order to avoid additional shifting around, which then might involve copying and injecting boxes as well as repackaging, two additional keys are available and these deal with the way boxes get anchored.

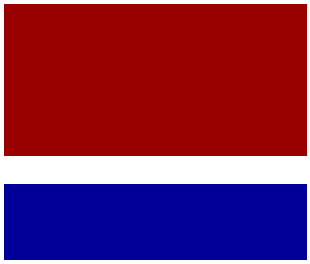
```
\vbox
    source 123
    \bgroup
        \offinterlineskip
        \blackrule[width=4cm,height=2cm,depth=0cm,color=darkred]\par
        \blackrule[width=4cm,height=0cm,depth=1cm,color=darkblue]\par
        \setbox\scratchboxtwo\hbox
            anchors "0004 "0001
          % anchor "00040001
            target 123
            orientation 1
            {\blackrule[width=2cm,height=1cm,depth=0cm,color=darkgreen]%
             \hskip-2cm
             \blackrule[width=2cm,height=0cm,depth=1cm,color=darkyellow]}%
          %
        \smash{\box\scratchboxtwo}%
    \egroup
```

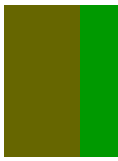
The anchor is just an number but with the plural keyword we can scan it as two because that is a bit easier on usage. The two numbers four byte numbers control the source to target anchoring and there is plenty room for future extensions because not all bits are used.

```
0x001  left origin
0x002  left height
0x003  left depth
0x004  right origin
0x005  right height
0x006  right depth
0x007  center origin
0x008  center height
0x009  center depth
0x00A  halfway total
0x00B  halfway height
0x00C  halfway depth
0x00D  halfway left
0x00E  halfway right
```

The target and source are handled in a way that sort of naturally binds them which involves a little juggling with dimensions in the backend. There is some additional control over this but usage is not advertized here because it might change.

One can set these anchoring related properties with keywords but there are also primitive box manipulators: `\boxanchor`, `\boxanchors`, `\boxsource` and `\boxtarget` that take a box number and value.





There are some helpers at the Lua end but I haven't completely made up my mind about them. Normally that evolves with usage.

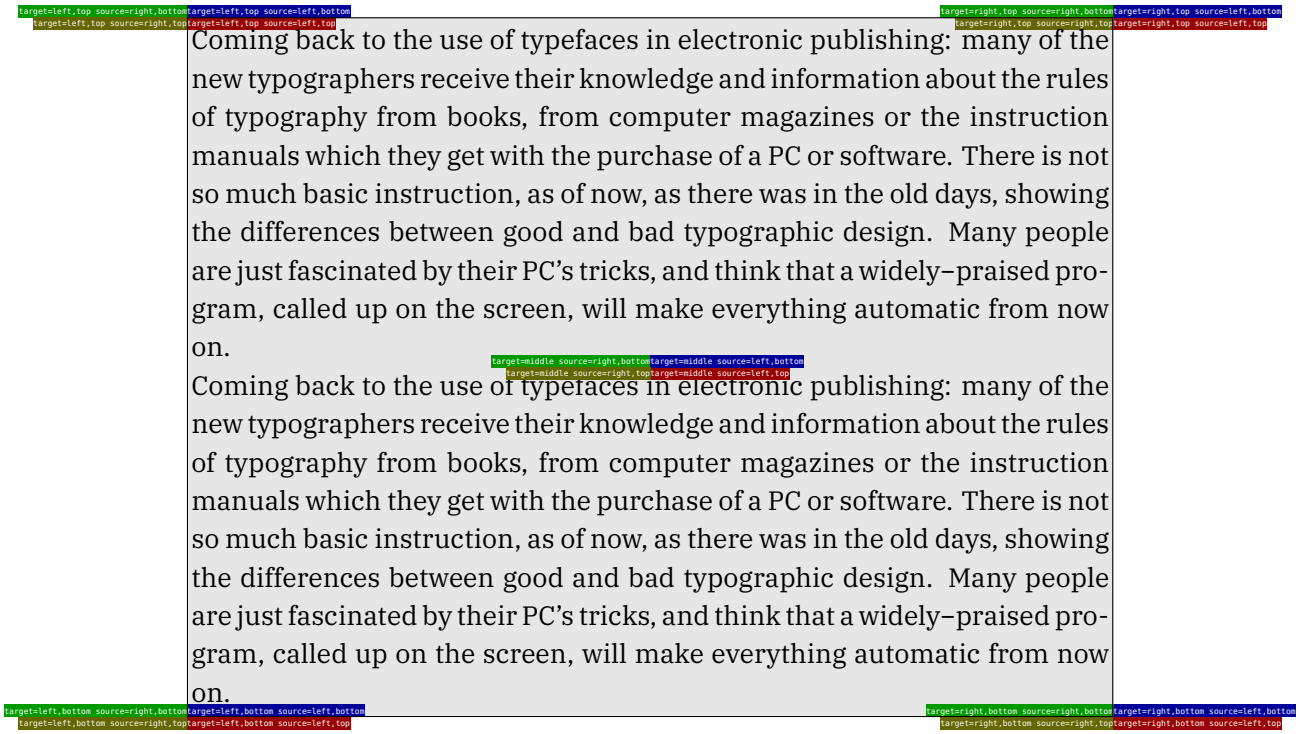
4.3 A first higher level interface

Exploring this here in more detail makes no sense because it is still experimental and also rather ConTEXt specific. As a teaser an interface that hooks into layers is shown:

```
\defineanchorboxoverlay[framed]

\def\DemoAnchor#1#2#3#4%
  {\setanchorbox
     [#1]%
     [target={#3},source={#4}]%
     \hbox{\backgroundline[#2]{\white\smallinfofont\setstrut\strut target=#3
                                                                  source=#4}}}

\def\DemoAnchorX#1#2%
  {\DemoAnchor{#1}{darkred}   {#2}{left,top}%
   \DemoAnchor{#1}{darkblue}  {#2}{left,bottom}%
   \DemoAnchor{#1}{darkgreen} {#2}{right,bottom}%
```

```
   \DemoAnchor{#1}{darkyellow}{#2}{right,top}%
   }%

\startsetups framed:demo
    \DemoAnchorX{framed:background}{left,top}%
    \DemoAnchorX{framed:background}{right,top}%
    \DemoAnchorX{framed:background}{left,bottom}%
    \DemoAnchorX{framed:background}{right,bottom}%
    \DemoAnchorX{framed:foreground}{middle}%
\stopsetups

\midaligned\bgroup
    \framed
      [align=normal,
       width=.7\textwidth,
       backgroundcolor=gray,
       background={color,framed:background,foreground,framed:foreground}]
      \bgroup
        \samplefile{zapf}\par
        \directsetup{framed:demo}%
        \samplefile{zapf}%
      \egroup
\egroup
```

Those familiar with ConTEXt will recognize the approach. This one basically is a more low level variant of layers and a high level variant of the primitives. Performance wise (in terms of memory usage and runtime) it sits in a sweet spot.



I played a bit with a mechanism that can store the embedded (to be anchored) content in a more independent way and it actually works okay. However, I'm not entirely sure if that solution is the best so for now it's commented. As usual it is also up to users to come up with demands.

5 A different approach to math spacing

Introduction

The TEX engine is famous for its rendering of math and even after decades there is no real contender. And so there also is no real pressure to see if we can do better. However, when Mikael Sundqvist ran into a Swedish math rendering specification and we started discussing a possible support for that in ConTEXt, it quickly became clear that the way TEX does spacing is a bit less flexible than one wishes for. We already have much of what is needed in place but it also has to work well with how TEX sees things:

1. Math is made from a sequence of atoms: a quantity with a nucleus, superscript subscript.[4] Atoms are spaced by `\thinmuskip`, `\medmuskip` and `\thickmuskip` or nothing, and that is sort of hard coded.
2. Atoms are organized by class and there are seven (or eight, depending on how you look at it) of them visible: binary symbols, relations, etc. The invisible ones, composites like fractions and fenced material (we call them molecules) are at some point mapped onto the core set. Molecules like fences have a different class left and right of the fenced material.
3. In addition the engine itself has all kind of spacing related parameters and these kick in automatically and sometimes have side effects. The same is true for penalties.

The normal approach to spacing other than imposed by the engine is to use correction space, like `\,` and I think that quite some TEX users think that this is how it is supposed to be. The standard way to enter math relates to scientific publishing and there the standards are often chiseled in stone so why should users tweak anyway. However, in ConTEXt we tend to start from the users and not the publishers end so there we can decide to follow different routes. Users can always work around something they don't like but we focus on reliable input giving predictable output. Also, when reading on, it is good to realize that it is all about the user experience here: it should look nice (which then of course makes one become aware of issues elsewhere) and we don't care much about specific demands of publishers in the scientific field: the fact that they often re-key content doesn't go well with users paying attention themselves, let alone the fact that nowadays they can demand word processor formats.

The three mentioned steps are fine for the average case but sometimes make no sense. It was definitely the best approach given time and resources but when LuaTEX went OpenType a lot of parameters were added and at that time we therefore added spacing by class pair. That not only decoupled the relation between the three (configurable) muskip parameters but also made it possible to use plenty of them. Now it must be said that for consistency having these three skips works great but given the tweaking expected from users consistency is not always what comes out.

This situation is very well comparable to the proclaimed qualities of the typesetting of text by TEX. Yes, it can do a great job, and often does, but users can mess up quite well. I remember that when we did tests with hz the outcomes were pretty unimpressive. When you give an audience a set of sample renderings, where each sample is slightly different and each user gets a randomized subset, the sudden lack of being able to compare (and agree) with another TEXie makes for interesting conclusions. They look for the opposites of what is claimed to be perfect. So, two lines with hyphens rate low, even if not doing it would look worse. The same for a few short words in the last line of a paragraph. Excessive spacing is also seen as bad. So, when asked why some paragraphs looked okay noticing (excessive and troublesome) expansion was not seen as a problem; instead it were hyphens that got the attraction.

The same is probably true for math: the input with lots of correction spaces or commands where characters would do can be horrible but it's just the way it is supposed to be. The therefore expected output can only

[4] I suddenly realize why in the engine noads have a nucleus field: they are atoms . . . but what does that make super and subscripts.

be perfect, right, independent of how one actually messed up spacing. But personally I think that it is often spacing messed up by users that make a TEX document recognizable. It compares to word processor results that one can sometimes identify by multiple consecutive spaces in the typeset text instead of using a glue model like TEX. Reaching perfection is not always trivial, but fortunately we can also find plenty of nice looking documents done with TEX.

The TEXbook has an excellent and intriguing chapter on the fine points of math and it definitely shows why Don Knuth wrote TEX as a tool for his books. He pays a lot of attention to detail and that is also why it all works out so well. If you need to render from unseen sources (as happens in an xml workflow) coming from several authors and have time nor money to check everything, you're off worse. And I'm not even talking of input where invisible Unicode spacing characters are injected. It is the TEX book(s) that has drawn me to this program and believe it or not, in the first project I was involved in that demanded typeset (quantum mechanics) math the ibm typewriter with changing bulbs ruled the scenery. In fact, our involvement was quickly cut off when we dared to show a chapter done in TEX that looked better.

Apart from an occasional tweak, in ConTEXt we never really used this opened up math atom pair spacing mechanism available in LuaTEX extensively. So, when I was pondering how to proceed it stroke me that it would make sense to generalize this mechanism. It was already possible (via a mode parameter) to bypass the second step mentioned above, but we definitely needed more than the visible classes that the engine had. In ConTEXt we already had more classes but those were meant for assigning characters and commands to specific math constructs (think of fences, fractions and radicals) so in the end they were not really classes. Considering this option was made easier by the fact that Mikael would do the testing and help configuring the defaults, which all will result in a new math user manual.

There are extensions introduced in LuaTEX and later LuaMetaTEX that are not discussed here. In this expose we concentrate on the features that were explored, extended and introduced while we worked on updating math support in LMTX.

An example

Before we go into details, let's give an example of unnoticed spacing effects. We use three simple formulas all using fractions:

```
\ruledhbox{$\frac{x^2}{a+1}$}
```

and:

```
\ruledhbox{$x + \frac{x^2}{a+1} = 10$}
```

as well as:

```
\ruledhbox{$\frac{1}{2}\frac{1}{2}x$}
```

$$\frac{x^2}{a+1} \qquad x + \frac{x^2}{a+1} = 10 \qquad \frac{1}{2}\frac{1}{2}x$$

If you look closely you see that the fraction has a little space at the left and right. Where does that come from? Because we normally don't put a tight frame around a fraction, we are not really aware of it. The spacing between what are called ordinary, operator, binary, relation and other classes of atoms is explained in the TEXbook (or "TEX by Topic" if you want a summary) and basically we have a class by class matrix that is built

into TEX. The engine looks at successive items and spacing depends on their (perceived) class. Because the number of classes is limited, and because the spacing pairs are hard coded, the engine cheats a little. Depending on what came before or comes next the class of an atom is adapted to suit the spacing matrix. One can say that a “reading mathematician” is built in. And most of the decisions are okay. If needed one can always wrap something in e.g. `\mathrel` but of course that also can interfere with grouping. All this is true for TEX, pdfTEX, XETEX and LuaTEX, but a bit different in LuaMetaTEX as we will see.

The little spacing on both edges of the fraction is a side effect of the way they are built internally: fractions are actually a generalized form of “stuff put on top of other stuff” and they can have left and/or right delimiters: this is driven by primitives that have names like `\atop` and `\atopwithdelims`. The way the components are placed is (especially in the case of OpenType) driven by lots of parameters and I will leave that out of the discussion.

When there are no delimiters, a so called `\nulldelimiterspace` will be injected. That parameter is set to 1.2 points and I have to admit that in ConTEXt I never considered letting that one adapt to the body font size, which means that, as we default to a 12 point body font, the value there should have been 1.44 points: mea culpa. When we set this parameter to zero point, we get this:

$$\frac{x^2}{a+1} \quad x+\frac{x^2}{a+1}=10 \quad \frac{1}{2}\frac{1}{2}x$$

As intermezzo and moment of contemplation I show some examples of fractions mixed into text. When we have the delimiter space set we get this:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$
test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$
test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$
test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$
test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$
test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$
test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

While with zero it looks like this, quite a different outcome:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$
test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$
test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$
test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$
test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$
test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$
test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

A little tracing shows it more clearly:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$
test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$
test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$
test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$
test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$

test$\frac{1}{82}$ test$\frac{1}{83}$ test$\frac{1}{84}$ test$\frac{1}{85}$ test$\frac{1}{86}$ test$\frac{1}{87}$ test$\frac{1}{88}$ test$\frac{1}{89}$ test$\frac{1}{90}$ test$\frac{1}{91}$ test$\frac{1}{92}$ test$\frac{1}{93}$ test$\frac{1}{94}$ test$\frac{1}{95}$ test$\frac{1}{96}$ test$\frac{1}{97}$ test$\frac{1}{98}$ test$\frac{1}{99}$ test$\frac{1}{100}$

You can zoom in and see where it interferes with margin alignment.

test$\frac{1}{1}$ test$\frac{1}{2}$ test$\frac{1}{3}$ test$\frac{1}{4}$ test$\frac{1}{5}$ test$\frac{1}{6}$ test$\frac{1}{7}$ test$\frac{1}{8}$ test$\frac{1}{9}$ test$\frac{1}{10}$ test$\frac{1}{11}$ test$\frac{1}{12}$ test$\frac{1}{13}$ test$\frac{1}{14}$ test$\frac{1}{15}$ test$\frac{1}{16}$ test$\frac{1}{17}$ test$\frac{1}{18}$ test$\frac{1}{19}$ test$\frac{1}{20}$ test$\frac{1}{21}$ test$\frac{1}{22}$ test$\frac{1}{23}$ test$\frac{1}{24}$ test$\frac{1}{25}$ test$\frac{1}{26}$ test$\frac{1}{27}$ test$\frac{1}{28}$ test$\frac{1}{29}$ test$\frac{1}{30}$ test$\frac{1}{31}$ test$\frac{1}{32}$ test$\frac{1}{33}$ test$\frac{1}{34}$ test$\frac{1}{35}$ test$\frac{1}{36}$ test$\frac{1}{37}$ test$\frac{1}{38}$ test$\frac{1}{39}$ test$\frac{1}{40}$ test$\frac{1}{41}$ test$\frac{1}{42}$ test$\frac{1}{43}$ test$\frac{1}{44}$ test$\frac{1}{45}$ test$\frac{1}{46}$ test$\frac{1}{47}$ test$\frac{1}{48}$ test$\frac{1}{49}$ test$\frac{1}{50}$ test$\frac{1}{51}$ test$\frac{1}{52}$ test$\frac{1}{53}$ test$\frac{1}{54}$ test$\frac{1}{55}$ test$\frac{1}{56}$ test$\frac{1}{57}$ test$\frac{1}{58}$ test$\frac{1}{59}$ test$\frac{1}{60}$ test$\frac{1}{61}$ test$\frac{1}{62}$ test$\frac{1}{63}$ test$\frac{1}{64}$ test$\frac{1}{65}$ test$\frac{1}{66}$ test$\frac{1}{67}$ test$\frac{1}{68}$ test$\frac{1}{69}$ test$\frac{1}{70}$ test$\frac{1}{71}$ test$\frac{1}{72}$ test$\frac{1}{73}$ test$\frac{1}{74}$ test$\frac{1}{75}$ test$\frac{1}{76}$ test$\frac{1}{77}$ test$\frac{1}{78}$ test$\frac{1}{79}$ test$\frac{1}{80}$ test$\frac{1}{81}$ test$\frac{1}{82}$ test$\frac{1}{83}$ test$\frac{1}{84}$ test$\frac{1}{85}$ test$\frac{1}{86}$ test$\frac{1}{87}$ test$\frac{1}{88}$ test$\frac{1}{89}$ test$\frac{1}{90}$ test$\frac{1}{91}$ test$\frac{1}{92}$ test$\frac{1}{93}$ test$\frac{1}{94}$ test$\frac{1}{95}$ test$\frac{1}{96}$ test$\frac{1}{97}$ test$\frac{1}{98}$ test$\frac{1}{99}$ test$\frac{1}{100}$

So, if you ever meet a user who claims perfection and superiority of typesetting, check out her/his work which might have inline fractions done the spacy way. It might make other visually typesetting claims less trustworthy. And yes, one can wonder if margin kerning could help here but as this content is wrapped in boxes it is unlikely to work out well (and not worth the effort).

In order to get a better picture of the spacing, two more renderings are shown. This time we show the bounding boxes of the characters too (you might need to zoom in to see it):

$$\frac{x^2}{a+1} \qquad x+\frac{x^2}{a+1}=10 \qquad \frac{1}{2}\frac{1}{2}x$$

Again we also show the zero case

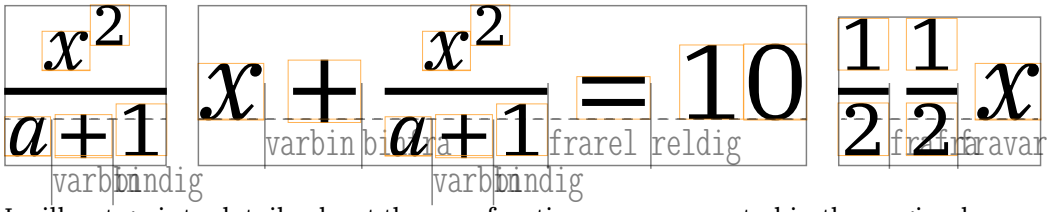
$$\frac{x^2}{a+1} \qquad x+\frac{x^2}{a+1}=10 \qquad \frac{1}{2}\frac{1}{2}x$$

This makes clear why there actually is this extra space around a fraction: regular operators have side bearings and thereby have some added space. And when we put a fraction in front of a symbol we need that little extra space. Of course a proper class pair spacing value could do the job but there is no fraction class. The engine cheats by changing the class depending on what follows or came before and this is why on the average it looks okay. However, these examples demonstrate that there are some assumptions with regard to for instance fonts and this is one of the reasons why the more or less official expected OpenType behavior as dictated by the Cambria font doesn't always work out well for fonts that evolved from the ones used in the TEX community. Also imagine how this interferes with the fact that traditional TEX fonts and the machinery do magic with cheating about width combined with italic correction (all plausible and quite clever but somewhat tricky with respect to OpenType).

Because here we discuss the way LuaMetaTEX and ConTEXt deal with this, the following examples show a probably unexpected outcome. Again first the non-zero case:

$$\frac{x^2}{a+1} \qquad x+\frac{x^2}{a+1}=10 \qquad \frac{1}{2}\frac{1}{2}x$$

And here the zero case:



I will not go into details about the way fractions are supported in the engine because some extensions are already around for quite a while. The main observation here is that in LuaMetaTEX we have alternative primitives that assume forward scanning, as if the numerator and denominator are arguments. The engine also supports skewed (vulgar) fractions natively where numerator and denominator are raised and lowered relative to the (often) slash. Many aspects of the rendering can be tuned in the so called font goodie files, which is also the place where we define the additional font parameters.

Atom spacing

If you are familiar with traditional TEX you know that there is some built in `ordbin` spacing. But there is no such pair for a fraction and a relation, simply because there is no fraction class. However, in LuaMetaTEX there is one, and we'd better set it up if we zero the margins of a fraction.

It is worth noticing that fractions are sort of special anyway. The official syntax is `n \over m` and numerator and denominator can be sub formulas. This is the one case where the parser sort of has to look back, which is tricky because the machinery is a forward looking one. Therefore, in order to get the expected styling (or avoid unexpected side effects) one will normally wrap all in braces as in: `{ {n} \over{m} }` which of course kind defeats the simple syntax which probably is supported for `1\over2` kind of usage, so a next challenge is to make `1/2` come out right. All this means that in practice we have wrappers like `\frac` which accidentally in LuaMetaTEX can be defined using forward looking primitives with plenty extra properties driven by keywords. It also means that fractions as expected by the engine due to wrapping actually can be a different kind of atom, which can have puzzling side effects with respect to spacing (because the remapping happens unseen).

Interesting is that adapting LuaMetaTEX to a more extensive model was quite doable, also because the code base had already been made more configurable. Of course it involved quite a bit of tedious editing and throwing out already nice and clean code that had taken some effort, but that's the way it is. Of course more classes also means that some storage properties had to be adapted within the available space but by sacrificing families that was possible. With 64 potential classes we now are back to 64 families compared to 7 classes and 256 families in LuaTEX and 7 classes and 16 families in traditional TEX.

Also interesting is that the new implementation is actually somewhat simpler and therefore the binary is a tad smaller too. But does all that mean that there were no pitfalls? Sure there were! It is worth noticing that doing all this reminded me of the early days of LuaTEX development, where Taco and I exchanged binaries and TEX code in a more or less constant way using Skype. For LuaMetaTEX we used good old mail for files and Mojca's build farm for binaries and Mikael and I spent many months exchanging information and testing out alternatives on a daily basis: it is in my opinion the only way to do this and it's fun too. It has been a lot of work but once we got going there was nothing that could stop us. A side effect was that there were no updates during this period, which was something users noticed.

In the spacing matrix there is `inner` and internally there's also some care to be taken of `vcenter`. The `inner` class is actually shared with the `variable` class which is not so much a real class but more a signal to the engine that when an alphabetic or numeric character is included it has to come from a specific family: upright family zero or math italic family one in traditional speak. But, what if we don't have that setup? Well,

then one has to make sure that this special class number is not associated (which is no big deal). It does mean that when we extend the repertoire of classes we cannot use slot seven. Always keep in mind that classes (and thereby signals) get assigned to characters (some defaults by the engine, others by the macro package). It is why in ConTEXt we use abstract class numbers, just in case the engine gets adapted.

We also cannot use slot eight because that one is a signal too: for a possible active math character, a feature somewhat complicated by the fact that it should not interfere with passing around such active characters in arguments. In math mode where we have lots of macros passing around content, this special class works around these side effects. We don't need this feature in ConTEXt because contrary to other macro packages we don't handle primes, pseudo superscripts potentially followed by other super and subscripts by making the `'` an active character and thereby a macro in math mode. This trickery again closely relates to preferable input, font properties, and limitations of memory and such at the time TEX showed up (much has to fit into 8, 16 or 32 bits, so there is not much room for e.g. more than 8 classes). Since we started with MkIV the way math is dealt with is a bit different than normally done in TEX anyway.

Atom rules

We can now control the spacing between every atom but unfortunately that is not good enough. Therefore, we arrive at yet another feature built into the engine: turning classes into other classes depending on neighbors. And this is precisely why we have certain classes. Let's quote "TEX by Topic": *The cases `*` (in the atom spacing matrix) cannot occur, because a `bin` object is converted to `ord` if it is the first in the list, preceded by `bin`, `op`, `open`, `punct`, `rel`, or followed by `close`, `punct`, or `rel`; also, a `rel` is converted to `ord` when it is followed by `close` or `punct`.*

We can of course keep these hard coded heuristics but can as well make that bit of code configurable, which we did. Below is demonstrated how one can set up the defaults at the TEX end. We use symbolic names for the classes.

```
\setmathatomrule  \mathbegincode          \mathbinarycode       % old
  \allmathstyles    \mathordinarycode       \mathordinarycode   % new

\setmathatomrule  \mathbinarycode         \mathbinarycode
  \allmathstyles    \mathbinarycode         \mathordinarycode
\setmathatomrule  \mathoperatorcode       \mathbinarycode
  \allmathstyles    \mathoperatorcode         \mathordinarycode
\setmathatomrule  \mathopencode           \mathbinarycode
  \allmathstyles    \mathopencode           \mathordinarycode
\setmathatomrule  \mathpunctuationcode    \mathbinarycode
  \allmathstyles    \mathpunctuationcode    \mathordinarycode
\setmathatomrule  \mathrelationcode       \mathbinarycode
  \allmathstyles    \mathrelationcode       \mathordinarycode

\setmathatomrule  \mathbinarycode         \mathclosecode
  \allmathstyles    \mathordinarycode       \mathclosecode
\setmathatomrule  \mathbinarycode         \mathpunctuationcode
  \allmathstyles    \mathordinarycode       \mathpunctuationcode
\setmathatomrule  \mathbinarycode         \mathrelationcode
  \allmathstyles    \mathordinarycode       \mathrelationcode

\setmathatomrule  \mathrelationcode       \mathclosecode
```

```
  \allmathstyles    \mathordinarycode      \mathclosecode
\setmathatomrule  \mathrelationcode      \mathpunctuationcode
  \allmathstyles    \mathordinarycode      \mathpunctuationcode
```

Watch the special class with \mathbegincode. This is actually class 62 so you don't need much fantasy to imagine that class 63 is \mathendcode, but that one is not yet used. In a similar fashion we can initialize the spacing itself:[5]

```
\setmathspacing \mathordcode   \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathordcode   \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathordcode   \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathordcode   \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathopcode    \mathordcode    \allmathstyles  \thinmuskip
\setmathspacing \mathopcode    \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathopcode    \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathopcode    \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathbincode   \mathordcode    \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathopcode     \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathopencode   \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathinnercode  \allsplitstyles \medmuskip

\setmathspacing \mathrelcode   \mathordcode    \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathopcode     \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathopencode   \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathinnercode  \allsplitstyles \thickmuskip

\setmathspacing \mathclosecode \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathclosecode \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathclosecode \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathclosecode \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathpunctcode \mathordcode    \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathopcode     \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathrelcode    \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathopencode   \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathclosecode  \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathpunctcode  \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathinnercode \mathordcode    \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathinnercode \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathinnercode \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathinnercode \mathopencode   \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathpunctcode  \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathinnercode  \allsplitstyles \thinmuskip
```

[5] Constant, engine specific, numbers like these are available in tables at the Lua end so we can change them and users can check that.

And because we have a few more atom classes this also needs to happen:

```
\letmathspacing  \mathactivecode   \mathordinarycode
\letmathspacing  \mathvariablecode \mathordinarycode
\letmathspacing  \mathovercode     \mathordinarycode
\letmathspacing  \mathundercode    \mathordinarycode
\letmathspacing  \mathfractioncode \mathordinarycode
\letmathspacing  \mathradicalcode  \mathordinarycode
\letmathspacing  \mathmiddlecode   \mathopencode
\letmathspacing  \mathaccentcode   \mathordinarycode

\letmathatomrule \mathactivecode   \mathordinarycode
\letmathatomrule \mathvariablecode \mathordinarycode
\letmathatomrule \mathovercode     \mathordinarycode
\letmathatomrule \mathundercode    \mathordinarycode
\letmathatomrule \mathfractioncode \mathordinarycode
\letmathatomrule \mathradicalcode  \mathordinarycode
\letmathatomrule \mathmiddlecode   \mathopencode
\letmathatomrule \mathaccentcode   \mathordinarycode
```

With `\resetmathspacing` we get an all-zero state but that might become more refined in the future. What is not clear from the above is that there is also an inheritance mechanism. The three special muskip registers are actually shortcuts so that changing the register value is reflected in the spacing. When a regular muskip value is (verbose or as register) that value is sort of frozen. However, the `\inherited` prefix will turn references to registers and constants into a delayed value: as with the predefined we now have a more dynamic behavior which means that we can for instance use reserved muskip registers as we can use the predefined. A bonus is that one can also use regular glue or dimensions, just in case one wants the same spacing in all styles (a muskip adapts to the size).

When you look at all of the above you might wonder how users are supposed to deal with math spacing. The answer is that often they can just assume that TEX does the right thing. If something somehow doesn't feel right, looking at solutions by others will probably lead a new user to just copy a trick, like injecting a `\thinmuskip`. But it can be that atoms depend on the already applied (or not) spacing, which in turn depends on values in the atom spacing matrix that probably only a few users have seen. So, in the end it all boils down to trust in the engine and one's eyesight combined with hopefully some consistency in adding space directives and often with TEX it is consistency that makes documents look right. In ConTEXt we have many more classes even if only a few characters fit in, like differential, exponential and imaginary.

Fractions again

We now return to the fraction molecule. With the mechanisms at our disposal we can change the fixed margins to more adaptive ones:

```
\inherited\setmathspacing \mathbinarycode   \mathfractioncode
    \allmathstyles \thickermuskip
\inherited\setmathspacing \mathfractioncode \mathbinarycode
    \allmathstyles \thickermuskip
\nulldelimiterspace\zeropoint
$x + \frac{1}{x+2} + x$
```

Here \thickermuskip is defined as 7mu plus 5mu where the stretch is the same as a \thickmuskip and the width 2mu more. We start out with three variants, where the last two have \nulldelimiterspace set to 0pt and the first one uses the 1.2pt.

$$x + \frac{1}{x+2} + x$$
$$x + \frac{1}{x+2} + x$$
$$x + \frac{1}{x+2} + x$$

When we now apply the new settings to the last one, and overlay them we get the following output: the first and last case are rather similar which is why this effort was started in the first place.

$$x + \frac{1}{x+2} + x$$

Of course these changes are not upward compatible but as they are tiny they are not that likely to change the number of lines in a paragraph. In display mode changes in horizontal dimensions also have little effect.

Penalties

An inline formula can be broken across lines, and for sure there are places where you don't want to break or prefer to break. In TEX line breaks can be influenced by using penalties. At the outer level of an inline math formula, we can have a specific penalty before and after a binary and/or relation. The defaults are such that there are no penalties set, but most macro packages set the so called \relpenalty and \binoppenalty (the op in this name does not relate to the operator class) so a value between zero and 1000. In LuaTEX we also have \pre variants of these, so we have four penalties that can be set, but that is not enough in our new approach.

These penalties are class bound and don't relate to styles, like atom spacing does. That means that while atom spacing involves 64 × 64 × 8 potential values, an amount that we can manage by using the discussed inheritance. The inheritance takes less values because which store 4 style values per class in one number. For penalties we only need to keep 64 × 2 in mind, plus a range of inheritance numbers. Therefore it was decided to also generalize penalties so that each class can have them. The magic commands are shown with some useless examples:

```
\letmathparent \mathdigitcode
    \mathbincode   % pre penalty
    \mathbincode   % post penalty
    \mathdigitcode % options
    \mathdigitcode % reserved
```

By default the penalties are on their own, like:

```
\letmathparent \mathdigitcode
  \mathdigitcode % pre penalty
  \mathdigitcode % post penalty
  \mathdigitcode % options
  \mathdigitcode % reserved
```

The options and reserved parent mapping are not (yet) discussed here. Unless values are assigned they are ignored.

```
\setmathprepenalty  \mathordcode 100
\setmathpostpenalty \mathordcode 600
\setmathprepenalty  \mathbincode 200
\setmathpostpenalty \mathbincode 700
\setmathprepenalty  \mathrelcode 300
\setmathpostpenalty \mathrelcode 800
```

As with spacing, when there is no known value, the parent will be consulted. An unset penalty has a value of 10000.

After discussing the implications of inline math crossing lines, Mikael and I decided there can be two solutions. Both can of course be implemented in Lua, but on the other hand, they make good extensions, also because it sort of standardized it. The first advanced control feature tweaks penalties:

```
\mathforwardpenalties  2  200 100
\mathbackwardpenalties 2  100  50
```

This will add 200 and 100 to the first two math related penalties, and 100 and 50 to the last two (watch out: the 100 will be assigned to the last one found, the 50 to the one before it). As with all things penalty and line break related, you need to have some awareness of how non-linear the badness calculation is as well of the fact that the tolerance and stretch related parameters play a role here.

The second tweak is setting `\maththreshold` to some value. When set to for instance `40pt`, formulas that take less space than this will be wrapped in a `\hbox` and thereby will never break across a page.[6] Actually that second tweak has a variant so we have three tweaks! Say that we have this sample formula wrapped in some bogus text and repeat that snippet a lot of times:

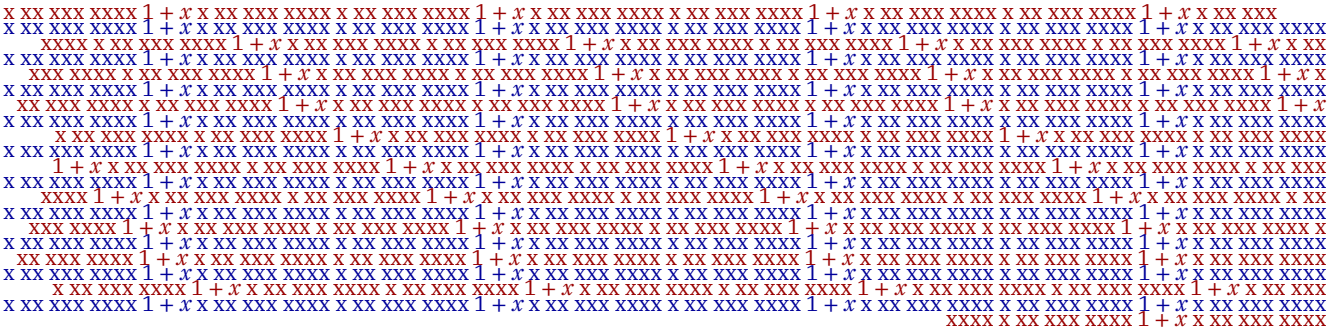
```
x xx xxx xxxx $1 + x$ x xx xxx xxxx
```

Now look at the example on the next page. You will notice that the red and blue text have different line breaks. This is because we have given the threshold some stretch and shrink. The red text has a zero threshold so it doesn't do any magic at all, while the second has this setup:

```
\setupmathematics[threshold=medium]
```

That setting set the threshold to `4em plus 0.75em minus 0.50em` and when the formula size exceeds the four quads the line break code will use the real formula width but with the given stretch and shrink. Eventually the calculated size will be used to repackage the formula. In the future we will also provide a way to define slack more relative to the size and/or number of atoms.

[6] A future version might inject severe penalties instead, time will learn.



x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
$1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx
x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx
xxxx x xx xxx xxxx $1 + x$ x xx xxx xxxx

Another way to influence line breaks is to use the two inline math related penalties that have been added at Mikael's suggestion:

```
\setupalign [verytolerant]
{\dorecurse{25}{test $\darkred   #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\preinlinepenalty  500 \postinlinepenalty -500
 \dorecurse{25}{test $\darkgreen #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\postinlinepenalty 500 \preinlinepenalty  -500
 \dorecurse{25}{test $\darkblue  #1^{#1} + x_{#1}^{#1}$ test }\blank}
```

To get an example that shows the effect takes a bit of trial and error because TeX does a very good job in line breaking. This is why we've set the tolerance and also use negative penalties.

In addition to the `\mathsurround` (kern) and `\mathsurroundskip` (glue) parameters this is a property of the nodes that mark the beginning and end of an inline math formula.

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test $7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$ test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$ test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$ test test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test $7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$ test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$ test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$ test test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test $7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$ test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$ test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$ test test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test

Flattening

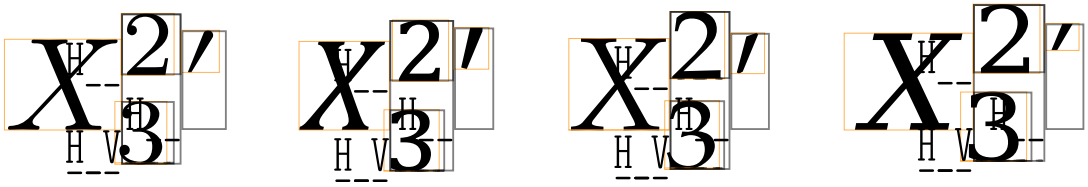
The traditional engine has some code for flattening math constructs that in the end are just one character. So in the end, `\tilde{u}` and `\tilde {uu}` become different objects even if both are in fact accents. In fact, when an accent is constructed there is a special code path for single characters so that script placement adapts to the shape of that character.

However because of interaction with primes, which themselves are sort of superscripts and due to the somewhat weird way fonts provide them when it comes to positioning and sizes, in ConTeXt we already are fooling

around a bit with these characters. For understandable reasons of memory usage, complexity and eightbitness primes are not a native TEX thing but more something that is handled at the macro level (although not in MkIV and LMTX).

In the end it was script placements on (widely) accented math characters that made us introduce a dedicated `\Umathprime` primitive that adds a prime to a math atom. It permits an uninterupted treatment of scripts while in the final assembly of the molecule the prime, superscript, subscript and maybe even prescripts that prime gets squeezed in. Because the concept of primes is missing in OpenType math an additional font parameter `PrimeTopRaisePercent` has been introduced as well as an `\Umathprimeraise` primitive. In retrospect I should have done that earlier but one tends to stick to the original as much as possible. However, at some point Mikael and I reached a state where we decided that proper (clean) engine extensions make way more sense than struggling with border cases and explaining users why things are so complicated.

The input `$ X \Uprimescript{'} ^2 _3 $` gives this:



Latin Modern Cambria Pagella Dejavu

With `\tracingmath = 1` this nicely traces as:

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord] family "0, character "58
.\superscript
..\mathchar[dig] family "0, character "32
.\subscript
..\mathchar[dig] family "0, character "32
.\primescript
..\mathchar[ord] family "0, character "27
```

Of course this feature can also be used for other prime like ornaments and who knows how it will evolve over time.

You can influence the positioning with `\Umathprimesupshift` which adds some kern between a prime and superscript. The `\Umathextraprimeshift` moves a prime up. The `\Umathprimeraise` is a font parameter that defaults to 25 which means a raise of 25%of the height. These are all (still) experimental parameters.

Fences

Fences can be good for headaches. Because the math that I (or actually my colleague) deal with is mostly school math encoded in presentation MathML (sort or predictable) or some form of sequential ascii based input (often rather messy and therefore unpredictable due to ambiguity) fences are a pain. A TEXie can make sure that left and right fences are matched. A TEXie also knows when something is an inline parenthesis or when a more high level structure is needed, for instance when parentheses have to scale with what they wrap. In that case the `\left` and `\right` mechanism is used. In arbitrary input missing one of those is

fatal. Therefore, handling of fences in ConTEXt is one of the more complex sub mechanisms: we not only need to scale when needed, but also catch asymmetrical usage.

A side effect of the encapsulating fencing construct is that it wraps the content in a so called inner (as in `\mathinner`) which means that we get a box, and it is a well known property of boxes that they don't break across lines. With respect to fences, a way out is to not really fence content, but do something like this:

```
\left(\strut\right. x + 1 \left.\strut\right)
```

and hope for the best. Both pairs are coupled in the sense that their sizes will match and the strut is what determines the size. So, as long as there is a proper match of struts all is well, but it is definitely a decent hack. The drawback is in the size of the strut: if a formula needs a higher one, larger struts have to be used. This is why in plain TEX we have these commands:

```
\def\bigl {\mathopen \big } \def\bigm {\mathrel\big } \def\bigr {\mathclose\big }
\def\Bigl {\mathopen \Big } \def\Bigm {\mathrel\Big } \def\Bigr {\mathclose\Big }
\def\biggl{\mathopen \bigg} \def\biggm{\mathrel\bigg} \def\biggr{\mathclose\bigg}
\def\Biggl{\mathopen \Bigg} \def\Biggm{\mathrel\Bigg} \def\Biggr{\mathclose\Bigg}

\def\big #1{{\hbox{$\left#1\vbox to  8.5pt{}\right.\nomathspacing$}}}
\def\Big #1{{\hbox{$\left#1\vbox to 11.5pt{}\right.\nomathspacing$}}}
\def\bigg#1{{\hbox{$\left#1\vbox to 14.5pt{}\right.\nomathspacing$}}}
\def\Bigg#1{{\hbox{$\left#1\vbox to 17.5pt{}\right.\nomathspacing$}}}

\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt} % renamed
```

The middle is kind of interesting because it has relation properties, while the `\middle` introduced in e-TEX got open properties, but we leave that aside.

In ConTEXt we have plenty of alternatives, including these commands, but they are defined differently. For instance they adapt to the font size. The hard coded point sizes in the plain TEX code relates to the font and steps available in there (either by next larger or by extensible). The values thereby need to be adapted to the chosen body font as well as the body font size. In MkIV and even better in LMTX we can actually consult the font and get more specific sizes.

But, this section is not about how to get these fixed sizes. Actually, the need to choose explicitly is not what we want, especially because TEX can size delimiters so well. So, take this code snippet:

```
$ x = \left( \dorecurse{40}{\frac{x}{x+#1} +} x \right) $
```

When we typeset this inline, as in $x = \left(\frac{x}{x+1} + \frac{x}{x+2} + \frac{x}{x+3} + \frac{x}{x+4} + \frac{x}{x+5} + \frac{x}{x+6} + \frac{x}{x+7} + \frac{x}{x+8} + \frac{x}{x+9} + \frac{x}{x+10} + \frac{x}{x+11} + \frac{x}{x+12} + \frac{x}{x+13} + \frac{x}{x+14} + \frac{x}{x+15} + \frac{x}{x+16} + \frac{x}{x+17} + \frac{x}{x+18} + \frac{x}{x+19} + \frac{x}{x+20} + \frac{x}{x+21} + \frac{x}{x+22} + \frac{x}{x+23} + \frac{x}{x+24} + \frac{x}{x+25} + \frac{x}{x+26} + \frac{x}{x+27} + \frac{x}{x+28} + \frac{x}{x+29} + \frac{x}{x+30} + \frac{x}{x+31} + \frac{x}{x+32} + \frac{x}{x+33} + \frac{x}{x+34} + \frac{x}{x+35} + \frac{x}{x+36} + \frac{x}{x+37} + \frac{x}{x+38} + \frac{x}{x+39} + \frac{x}{x+40} + x\right)$, we get nicely scaled fences but in a way that permits line breaks. The reason is that the engine has been extended with a `fenced` class so that we can recognize later on, when TEX comes to injecting spaces and penalties, that we need to unpack the construct. It is another beneficial side effect of the generalization.

The Plain TEX code can be used to illustrate some of what we discussed before about fractions. In the next code we use excessive delimiter spacing:

```
\def\Bigg#1{%  watch the wrapping in a box
  {%
    \hbox {%
        $\normalleft#1\vbox to 17.5pt{}\normalright.\nomathspacing$%
    }%
```

```
  }%
}

\nulldelimiterspace0pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par
```

This renders as follows. We explicitly set `\nulldelimiterspace` to values because in ConTEXt it is now zero by default.

0pt with reset at end

10pt with reset at end

10pt without reset at end

Radicals

In traditional TEX a radical with degree is defined as macro. That macro does some measurements and typesets the result in four sizes for a choice. The macro typesets the degree in a box that contains the degree as formula. There is a less guesswork going on than with respect to how the radical symbol is shaped but as we're talking plain TEX here it works out okay because the default font is well known.

Radicals are a nice example of a two dimensional 'extender' but only the vertical dimension uses the extension mechanism, which itself operates either horizontally or vertically, although in principle it could go both ways. The horizontal extension is a rule and the fact that the shape is below the baseline (as are other large symbols) will make the rule connect well: the radical shape sticks out a little, so one can think of the height reflecting the rule height.[7] In OpenType fonts there is a parameter and in LuaTEX we use the default rule thickness for traditional fonts, which is correct for Latin Modern. There are more places in the fonts where the design relates to this thickness, for instance fraction rules are supposed to match the minus, but this is a bit erratic if you compare fonts. This is one of the corrections we apply in the goodie files.

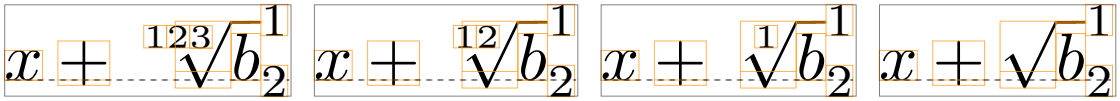
In OpenType the specification of the radical also includes spacing properties of the degree and that is why we have a primitive in LuaTEX that also handles the degree. It is what we used in ConTEXt MkIV. But ... we actually end up with a situation that compares to the already discussed fraction: there is space added before a radical when there is a degree. However, because we now have a radical atom class, we can avoid using that one and use the new pairwise spacing. Some fuzzy spacing logic in the engine could

[7] When you zoom in you will notice that this is not always optimal because of the way the slope touched the rule.

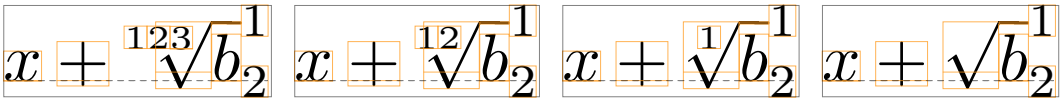
therefore be removed and we assume that \Umathradicaldegreebefore is zero. For the record: the \Umathradicaldegreeafter sort of tells how much space there is above the low part of the root, which means that we can compensate for multi-digit degrees.

Zeroing a parameter is something that relates to a font which means that it has to happen for each math font which in turn can mean a family-style combination. In order to avoid that complication (or better: to avoid tracing clutter) we have a way to disable a parameter:

```
\ruledhbox{$x + \sqrt[123]{b}^1_2$}
\ruledhbox{$x + \sqrt[12] {b}^1_2$}
\ruledhbox{$x + \sqrt[1]  {b}^1_2$}
\ruledhbox{$x + \sqrt     {b}^1_2$}
```



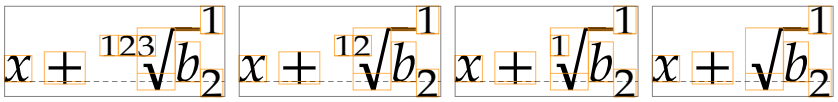
```
\setmathignore\Umathradicaldegreebefore 0
```



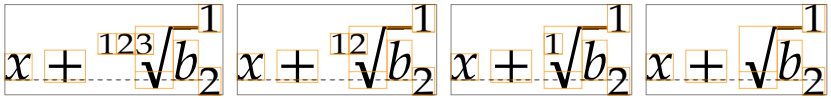
```
\setmathignore\Umathradicaldegreebefore 1
```

Latin Modern

One problem with these spacing parameters is that they are inconsistent across fonts. The Latin Modern has a rather large space before the degree, while Cambria and Pagella have little. That means that when you prototype a mechanism the chosen solution can look great but not so much when at some point you use another font.



```
\setmathignore\Umathradicaldegreebefore 0
```



```
\setmathignore\Umathradicaldegreebefore 1
```

Cambria

More fences

One of the reasons why the MkII and MkIV fence related mechanism is somewhat complex is that we want a clean solution for filtering fences like parenthesis by size, something that in the traditional happens via a fake fence pair that encapsulates a strut of a certain size. In LMTX we use the same approach but have made the sequence more configurable. In practice that means that the values 1 up to 4 are just that but for some fonts we use the sequence `1 3 5 7`. There was no need to adapt the engine as it already worked quite well.

Integrals

The Latin Modern fonts have only one size of big operators and one reason can be that there is no need for more. Another reason can be that there was just no space in the font. However, an OpenType font has plenty slots available and the reference font Cambria has integral signs in sizes as well as extensibles.

In LuaTEX we already have generic vertical extensibles but that only works well with specified sizes. And, cheating with delimiters has the side effect that we get the wrong spacing. In LuaMetaTEX however we have ways to adapt the size to what came or what comes. In fact, it is a mechanism that is available for any atom that we support. However, it doesn't play well with script and this whole `\limits` and `\nolimits` is a bit clumsy so Mikael and I decided that different route had to be followed. For adaptive large operators we provide this interface:

```
$ x + \integral [color=darkred,top={t},bottom={b}] {\frac{1}{x}} = 10 $

$ x + \startintegral [color=darkblue,top={t},bottom={b}]
    \frac{1}{x}
\stopintegral = 10 $

$ x + \startintegral [color=darkgreen,top={t},bottom={b},method=vertical]
    \frac{1}{x}
\stopintegral= 10 $
```

This text is not about the user interface so we won't discuss how to define additional large operators using one-liners.

$$x + \int_b^t \frac{1}{x} = 10 \quad x + \int_b^t \frac{1}{x} = 10 \quad x + \int_b^t \frac{1}{x} = 10$$

The low level LuaMetaTEX implementation handles this input:

```
\Uoperator          \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator   limits \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator nolimits \Udelimiter "0 \fam "222B {top} {bottom} {...}
```

plus the usual keywords that fenced accept, because after all, this is just a special case of fencing.

Currently these special left operators are implemented as a special case of fences because that mechanism does the scaling. It means that we need a (bogus) right fence, or need to brace the content (basically create an atom). When no right fence is found one is added automatically. Because there is no real fencing, right fences are removed when processing takes place. When you specify a `class` that one will be used for the left and right spacing, otherwise we have open/close spacing.

Going details

When the next feature was explored Mikael tagged it as math micro typography and the reason is that you need not only to set up the engine for it but also need to be aware of this kind of spacing. Because we wanted to get rid of this script spacing that the font imposes we configured ConTEXt with:

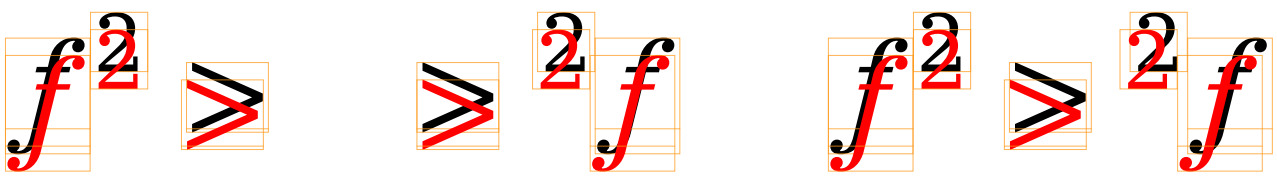
```
\setmathignore\Umathspacebeforescript\plusone
\setmathignore\Umathspaceafterscript \plusone
```

This basically nils all these tiny spaces. But the latest configuration has this instead:

```
% \setmathignore \Umathspacebeforescript\zerocount % default
% \setmathignore \Umathspaceafterscript \zerocount % default

\mathslackmode \plusone

\setmathoptions\mathopcode      \plusthree
\setmathoptions\mathbinarycode  \plusthree
\setmathoptions\mathrelationcode\plusthree
\setmathoptions\mathopencode    \plusthree
\setmathoptions\mathclosecode   \plusthree
\setmathoptions\mathpunctcode   \plusthree
```

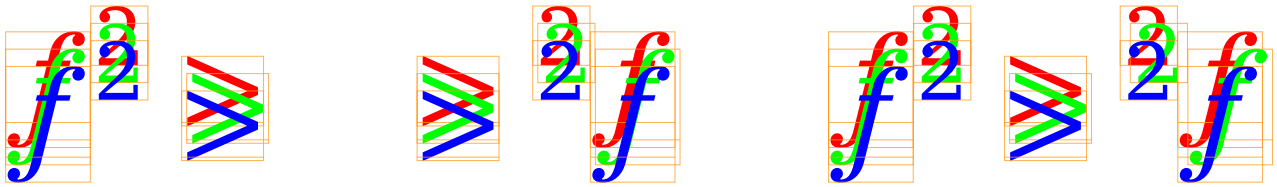
This tells the engine to convert these spaces into what we call slack: disposable kerns at the edges. But it also converts these kerns into a glue component when possible. As with all these extensions it complicates the machinery but users will never see that. Now, the last six lines do the magic that made us return to honoring the spaces: we can tell the engine to ignore this slack when there are specific classes at the edges. These options are a bitset and 1 means “no slack left” and 2 means “no slack right” so 3 sets both.

```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\zerocount
      \hbox\bgroup
         \setmathignore\Umathspacebeforescript\zerocount
         \setmathignore\Umathspaceafterscript \zerocount
         #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\plusone
        \setmathignore\Umathspaceafterscript \plusone
         #1
      \egroup
   \egroup}

\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >      $}} {}
    {\showglyphs\TestSlack{$    > f^^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^^2$}} {}
\stopcombination
```



Because this overall removal of slack is not granular enough a while later we introduced a way to set this per class, as is demonstrated in the following example.

```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\plusone
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\zerocount
        \setmathignore\Umathspaceafterscript \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\green
        \setmathoptions\mathrelationcode \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\blue
        \setmathoptions\mathrelationcode \plusthree
        #1
      \egroup
   \egroup}

\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >      $}} {}
    {\showglyphs\TestSlack{$    > f^^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^^2$}} {}
\stopcombination
```



Of course we need to experiment a lot with real documents and it might take a while before all this is stable (in the engine and in ConTEXt). And as we don't need to conform to the decades old golden TEX math standards we have some degrees of freedom in this: for Mikael and me it is pretty much a visual thing where we look closely at large samples. Of course in practice details get lost when we print at 10 point but that doesn't mean we can't provide the best experience.[8]

As we mention class specific options, we also need to mention the special case where we have for instance simple formulas like single atoms (for instance digits) are preceded by a sign (binary). These special spacing cases are handled by a lookahead flag that can be set `\setmathoptions <class>`, like the slack flags. More options might become available in due time. When set the lookahead will check for the automatically injected end class atom and use that for spacing when found. The mentioned lookahead is one of the hard coded heuristics in the traditional engine but here we need to explicitly configure it.

[8] Whenever I look at (my) old (math) school books I realize that Don Knuth had very good reasons to come up with TEX and, it being hard to beat, TEX still sets the standard!

Ghosts

As plain TEX has macros like \vphantom you also find them in macro packages that came later. These create a boxes that have their content removed after the dimensions are set. They take space and are invisible but there's also nothing there.

A variant in the upgraded math machinery are ghosts: these are visible in the sense that they show up but ignored when it comes to spacing. Here is an example. The option bit set here tells the engine that we ghost at the right, so we have ghosts around the relation (it controls where the spacing ends up).

```
$
  x
  \mathatom class \mathghostcode                  {!!}
  >
  \mathatom class \mathghostcode options "00000020 {!!}
  1
  \quad
  x
  \mathatom class \mathghostcode                  {\hbox{\smallinfofont ord}}
  >
  \mathatom class \mathghostcode options "00000020 {\hbox{\smallinfofont dig}}
  1
$
```

You never know when this comes in handy but it fits in the new, more granular approach to spacing. The code above shows that it's just a class, this time with number 18.

Struts

In order to get consistent spacing the ConTEXt macro package makes extensive use of struts in text mode as well as math mode. The normal way to implement that is either an empty box or a zero width rule, both with a specifically set height and depth. In ConTEXt MkII and MkIV (and for a long time in LMTX too) they were rules so that we could visualize them: they get some width and kerns around them to compensate for that.

In order to not let them interfere with spacing we could wrap them into a ghost atom but it is kind of ugly. Anyway, before we had these ghost atoms an alternative to struts was already implemented: a special kind of rule. The reason is that I wanted a cleaner and more predictable way to adapt struts to the math style uses and sometimes predicting that is fragile. What we want is a delayed assignment of dimensions.

We have two solutions. The first one uses two math parameters that themselves adapt to the style, as do other parameters: \Umathruleheight and \Umathruledepth. The other solution relates a font (or family) and character with the strut rule which is then used as measure for the height and depth. Just for the record: this also works in text mode, which is why a recent LMTX also does use that for struts now. The optional visualization is just part of the regular visualization mechanism in ConTEXt which already had provisions for struts. A side effect of this is that the rule primitives now accept three more keywords: font, fam and char, in addition to the already present traditional ones width, height and depth, the (backend) margin ones left (or top) and right (or bottom) options, as well as xoffset and yoffset). The command that

creates a rule with subtype `strut` is simply `\srule`. Because struts are rather macro package specific I leave it to this.

One positive side effect is that we could simplify the ConTEXt fraction mechanism a bit. Over time control over the (font driven) gaps was introduced but that is not really needed because we zero the gaps anyway. There was also a tolerance mechanism which again was not used. However, for skewed fractions we do use the new tolerance mechanism as well as gap control.

Atoms

Now that we have generic atoms (`\mathatom`) another, sometimes confusing aspect of the math parsing can be solved. Take this:

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ \MyBin _ \MyBin $
```

The parser just doesn't like that which means that one has to use

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ {\MyBin} _ {\MyBin} $
```

or:

```
\def\MyBin{{\mathbin{\tt mybin}}}
$ x ^ \MyBin _ \MyBin $
```

But the later has side effects: it creates a list that can influence spacing. It is for that reason that we do accept atoms where they were not accepted before. Of course that itself can have side effects but at least we don't get an error message. It fits well into the additional (user) classes model. And, given that in ConTEXt the `\frac` command is actually wrapped as `\mathfrac` the next will work too:

```
$ x^\frac{1}{2} + x^{\frac{1}{2}} $
```

but in practice you should probably use the braced version here for clarity.

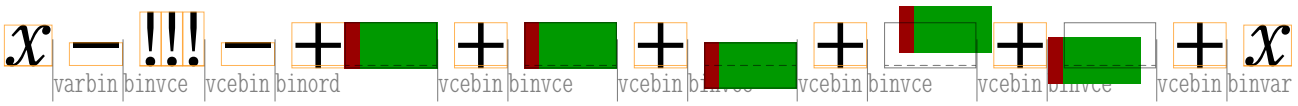
The `vcenter` primitive

Traditionally this primitive is bound to math but it had already been adapted to also work in text mode. As part of the upgrade of math we can now also pass all the options that normal boxed take and we can also cheat with the axis. Here is an example:

```
\def\TEST{\hbox\bgroup
    \darkred   \vrule width  2pt height 4pt
    \darkgreen \vrule width 10pt depth  2pt
\egroup}
$
    x  - \mathatom \mathvcentercode {!!!} -
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter axis 1                   {\TEST}
    + \ruledvcenter xoffset  2pt yoffset  2pt {\TEST}
    + \ruledvcenter xoffset -2pt yoffset -2pt {\TEST}
```

```
    + x
$
```

There was already a vcenter class available before we did this:



Text

Sometimes you want text in math, for instance `sin` or `cos` but text in math is not really text:

```
$ \setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu fin(x) $
```

The result demonstrates that what looks like a word actually becomes three math atoms:

Okay, so how about then wrapping it into a text box:

```
$
    \setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu
    fin(x) \quad \hbox{fin}(x)
$
```

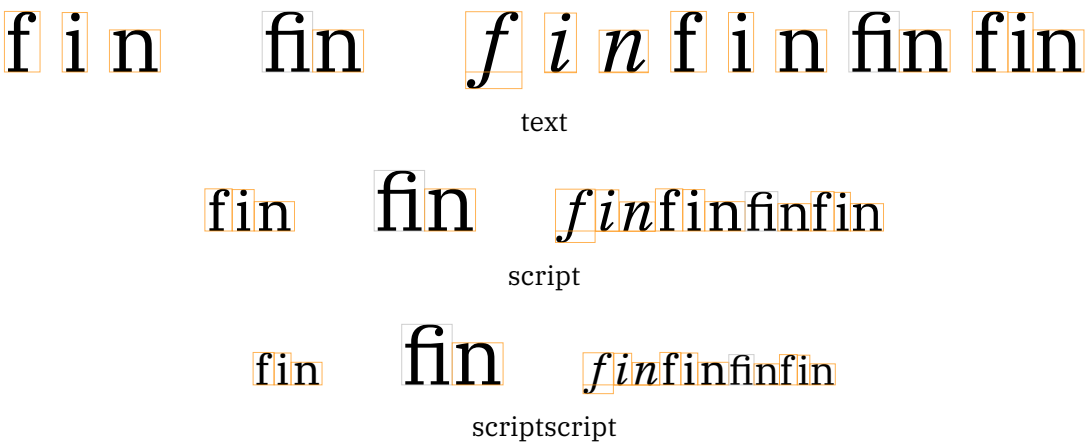
Here we get:

We even get a ligature which might be an indication that we're not using a math font which indeed is the case: the box is typeset in the regular text font.

```
\def\Test#1%
  {\setmathspacing\mathordinarycode\mathordinarycode\textstyle 5mu
   $\showglyphs
   #1% style
   {\tf fin} \quad
   \hbox{fin} \quad
   \mathatom class \mathordinarycode textfont {fin}
   \mathatom class \mathordinarycode textfont {\tf fin}
   \mathatom class \mathordinarycode textfont {\hbox{fin}}
   \mathatom class \mathordinarycode mathfont {\hbox{fin}}
```

```
$}
```

When we feed this macro with the `\textstyle`, `\scriptstyle` and `\scriptscriptstyle` we get:



Here you see a new atom option action: `textfont` which does as much as setting the font to the current family font and the size to the one used in the style. For the record: you only get ligatures when they are configured and provided by the font (and as math is a script itself it is unlikely to work).[9]

Tracing

I won't discuss the tracing features in ConTEXt here but for sure the visualizer helps a lot in figuring out all this. In LuaMetaTEX we carry a bit more information with the resulting nodes so we can provide more details, for instance about the applied spacing and penalties. Some is shown in the examples. A more recent tracing feature is this:

```
\tracingmath   1
\tracingonline 1
$
    \mathord (
    \mathord {(}
    \mathord \Udelimiter"4 0 `(
    \Udelimiter"4 0 `(
$
```

That gives us on the console (the dots represent detailed attribute info that we omit here):

```
7:3: > \inlinemath=
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathlist
7:3: ...\noad[open][...]
7:3: ....\nucleus
7:3: .....\mathchar[open] family "0, character "28
```

[9] The existing mechanisms in ConTEXt already dealt with this but it is nevertheless nice to have it as a clean engine feature.

```
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[open][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
```

A tracing level of 2 will spit out some information about applied spacing and penalties between atoms (when set) and level 3 will show the math list before the first and second pass (a mix of nodes and noads) we well as the result (nodes) plus return some details about rules, spacing and penalties applied.

Is there more?

The engine already provides the option to circumvent the side effect of a change in a parameter acting sort of global: the last value given is also the one that a second pass starts with. The `\frozen` prefix will turn settings into local ones but that's another (already old) topic. There are many such improvements and options not mentioned here but you can find them mentioned and explained in older development stories. A lot has been around for a while but not been applied in ConTEXt yet.

When TEX was written one important property (likely related to memory consumption) is that node lists have only forward pointers. That means that the state of preceding material has to be kept track of: there is no going (or looking) back. In LuaTEX we have double linked lists so in principle we can try to be more clever but so far I decided not to touch the math machinery in that way. But who knows what comes next.

Those italics

Right from the start of LuaTEX it became clear that the fact that TEX assumes the actual width of glyphs to be incremented by the italic correction that then selectively is removed has been an issue. It made for successive attempts to improve spacing in ConTEXt by providing pseudo features. But, when we moved from assembled Unicode math fonts to 'real' ones that became messy: what trick to apply when and even worse where? In the end there are only a very few shapes that actually are affected in the sense that when we don't deal with them it looks bad. It also happens that one of those shapes is the italic 'f', a letter that is used frequently in math. It might even be safe to say that the simple fact that the math italic f has this excessively wrong width and thereby pretty large italic correction is the cause of many problems.

In the LMTX approach Mikael and I settled on patching shapes in the so called font goodie files, aka `lfg` files and only a handful of entries needed a treatment. This makes a good case for removing the traditional font code path from LuaMetaTEX.

`modern`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`cambria`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`pagella`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`termes`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

bonum: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

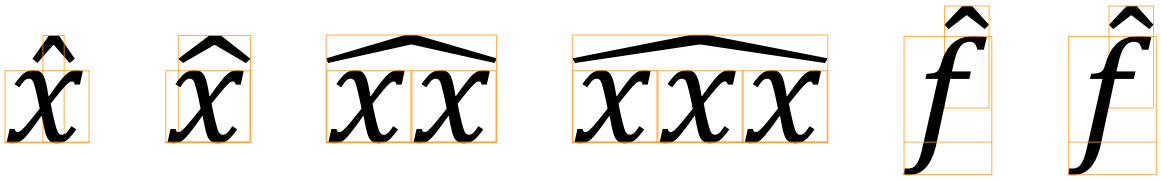
One of the other very sloped symbol is the integral, although most fonts have them more upright than tex has. Of course there are many variants of these integrals in a math font. Here we also have some font parameters that we can tune, which is what we do.

Accents

Accents are common in languages other than English and it's English that TEX was made for. Although the seven bit variant became eight bit handling accents never was sophisticated and one of the main reasons is of course that one could use pre-built composed characters. The OpenType format brought proper anchoring (aka marks) to font formats and when LuaTEX deals with text those kick in. In OpenType math however, anchoring is kind of limited to the top position only. Because the TEX Gyre fonts are based on traditional TEX fonts, their accents have not become better suited.

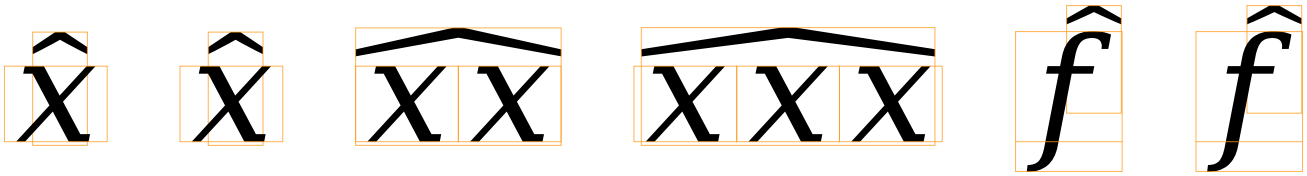
```
$ \hat{x} \enspace \widehat{x} \enspace \widehat{xx} \enspace \widehat{xxx}
  \enspace \hat{f} \enspace \widehat{f} $
```

When looking at examples you need to be aware of the fact hat fonts can have been adapted in the goodie files.[10] So, for instance bounding boxes and such can differ from the original. Anyway, the previous code in Cambria looks as follows.



With Latin Modern we get:

And Dejavu comes out as:



As you can see there are some differences. In for instance Latin Modern the shape of the hat and smallest wide hat are different and the first wide one has zero dimensions combined with a negative anchor. When an accented character is followed by a superscript or prime the italic correction of the base kicks in but that cannot be enough to not let this small wide hat overflow into the script. We could compensate for it but then we need to know the dimensions. Of course we can consult the bounding box but it makes no sense to let heuristics enter the machinery here while we're in the process generalization. One option is to have two

[10] Extreme examples can be found for Lucida Bright where we not only have to fix the extensible parts of horizontal braces but also have to provide horizontal brackets.

extra parameters that can be used when the width of the accent comes close to the width of the base (we then assume that zero accent width means that it has base width) we add an additional kern. In the end we settled for a (semi automatic) correction option in the goodie files.

There are actually three categories of extensible accents to consider: those that resemble the ones used in text (like tildes and hats), those wrapping something (like braces and bracket but also arrows) and rules (that in traditional TEX indeed are rules). In ConTEXt we have different interfaces for each of these in order to have a more extensive control. The text related ones are the simplest and closest to what the engine supports out of the box but even there we use tweaked glyphs to get better spacing because (of course) fonts have different and inconsistent spacing in the boundingbox above and below the real shape. This is again some tweak that we moved from being *automatic* to being *under goodie file control*. But this is all too ConTEXt specific to discuss here in more detail.

Decision time

After lots of tests Mikael and I came to the conclusion that we're facing the following situation. When typesetting math most single characters are italic and we already knew from the start of the LuaTEX project that the italics shapes are problematic when it comes to typesetting math. But it looks like even some upright characters can have italic correction: in TexGyreBonum for instance the bold upright `f` has italic correction, probably because it then can (somehow) kern with a following `i`. It anyhow assumes no italic correction to be applied between these characters.

In the end the mixed math font model model got more and more stressed so one decision was to simply assume fonts to be used that are either Cambria like OpenType, or mostly traditional in metrics, or a hybrid of both. It then made more sense to change the engine control options that we have into ones that simply enable certain code paths, independent of the fact if a font is OpenType or not. It then become a bit "crap in, crap out", but because we already tweak fonts in the goodie files it's quite okay. Some fonts have bad metrics anyway or miss characters and it makes no sense to support abandoned fonts either. Also, when a traditional font is assembled one can set up the engine with different flags and we can deal with it as we wish. In the end it is all up to the macro package to configure things right, which is what we tried to do for months when rooting out all the artifacts that fonts bring.[11]

That said, the reason why some (fuzzy) mixed model works out okay (also in LuaTEX) is that proper OpenType fonts use staircase kerns instead of italic correction. They also have no ligatures and kerns. We also suspect that not that much attention is paid to the rendering. It's a bit like these "How many f's do you count in this sentence?" tests where people tend to overlook `of`, `if` and similar short words. Mathematicians loves `f`'s but probably also overlook the occasionally weird spacing and kerning.

A side effect is that mixing OpenType and traditional fonts is also no longer assumed which in turn made a few (newly introduced) state variables obsolete. Once everything is stable (including extensions discussed before) some further cleanup can happen. Another side effect is that one needs to tell the engine what to apply and where, like this:

```
\mathfontcontrol\numexpr \zerocount
   +\overrulemathcontrolcode
   +\underrulemathcontrolcode
   +\fractionrulemathcontrolcode
   +\radicalrulemathcontrolcode
   +\accentskewhalfmathcontrolcode
```

[11] In previous versions one could configure this per font but that has been dropped.

```
   +\accentskewapplymathcontrolcode
 % + checkligatureandkernmathcontrolcode
   +\applyverticalitalickernmathcontrolcode
   +\applyordinaryitalickernmathcontrolcode
   +\staircasekernmathcontrolcode
 % +\applycharitalickernmathcontrolcode
 % +\reboxcharitalickernmathcontrolcode
   +\applyboxeditalickernmathcontrolcode
   +\applytextitalickernmathcontrolcode
   +\checktextitalickernmathcontrolcode
 % +\checkspaceitalickernmathcontrolcode
   +\applyscriptitalickernmathcontrolcode
   +\italicshapekernmathcontrolcode
\relax
```

There might be more control options (also for tracing purposes) and some of the symbolic (ConTEXt) names might change for the better. As usual it will take some years before all is stable but because most users use the latest greatest version it will be tested well.

After this was decided and effective I also decided to drop the mapping from traditional font parameters to the OpenType derives engine ones: we now assume that the latter ones are set. After all, we already did that in ConTEXt for the virtual assemblies that we started out with in the beginning of LuaTEX and MkIV.

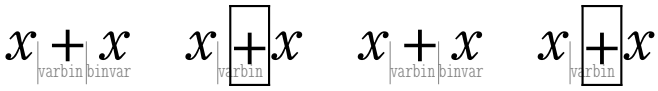
Dirty tricks

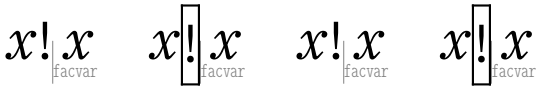
Once you start playing with edge cases you also start wondering if some otherwise complex things can be done easier. The next macro brings together a couple of features discussed in previous sections. It also uses two state variables: `\lastleftclass` and `\lastrightclass` that hold the most recent edge classes.

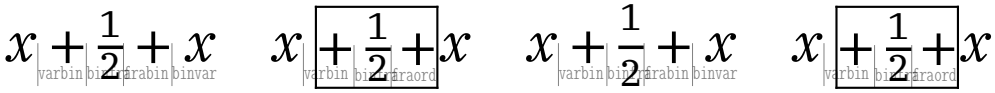
```
\tolerant\permanent\protected\def\NiceHack[#1]#:#2% special argument parsing
  {\begingroup
   \setmathatomrule
     \mathbegincode\mathbincode % context constants
     \allmathstyles
     \mathbegincode\mathbincode
   \normalexpanded
     {\setbox\scratchbox\hpack
        ymove \Umathaxis\Ustyle\mathstyle % an additional box property
        \bgroup
          \framed % a context macro
            [location=middle,#1]
            {$\Ustyle\mathstyle#2$}%
        \egroup}%
   \mathatom
     class 32 % an unused class
     \ifnum\lastleftclass <\zerocount\else leftclass  \lastleftclass\fi
     \ifnum\lastrightclass<\zerocount\else rightclass \lastrightclass\fi
     \bgroup
       \box\scratchbox
     \egroup
```

```
   \endgroup}

\def\MyTest#1%
  {$              x #1                          x $\quad
   $              x \NiceHack[offset=0pt]{#1} x $\quad
   $\displaystyle x #1                          x $\quad
   $\displaystyle x \NiceHack[offset=0pt]{#1} x $}

\scale[scale=2000]{\MyTest{>}}            \blank
\scale[scale=2000]{\MyTest{+}}            \blank
\scale[scale=2000]{\MyTest{!}}            \blank
\scale[scale=2000]{\MyTest{+\frac{1}{2}+}}\blank
\scale[scale=2000]{\MyTest{\frac{1}{2}}}  \blank
```

Of course this is not code you immediately come up with after reading this text, also because you need to know a bit of ConTEXt.



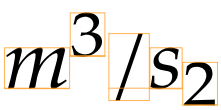




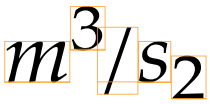
There are a few control options, like `\noatomruling` that can be used to prevent rules being applied to the next atom. We can use these in order to achieve more advanced alignment results, but discussing math alignments would demand many more pages than make sense here.

Tuned kerning

The ConTEXt distribution has dedicated code for typesetting units that dates back to the mid nineties of the previous century but was (code wise) upgraded from MkII to MkIV which made it end up in the physics name space. There is not much reason to redo that code but when we talk new spacing classes it might make sense at some point to see if we can use less code for spacing by using a 'unit' class. When Mikael pointed out that, for instance in Pagella:



doesn't space well the obvious answer is: use the units mechanism because this kind of rendering was why it was made in the first place. However, the question is of course, can we do better anyway. The chosen solution uses a combination of class options and tweaked shape kerning:



An example of a class setup in ConTEXt is:

```
\setmathoptions\mathdivisioncode\numexpr
    \nopreslackclassoptioncode     +\nopostslackclassoptioncode
   +\lefttopkernclassoptioncode    +\righttopkernclassoptioncode
   +\leftbottomkernclassoptioncode +\rightbottomkernclassoptioncode
\relax
```

and, although we don't go into the details of tweaking here, this is the kind if code you will find in the goodie file:

```
{
    tweak = "kerns",
    list  = {
        [0x2F] = {
            topleft     = -0.3,
            bottomright =  0.2,
        }
    }
}
```

where the numbers are a percentage of the width. This specification translates in a math staircase kerning recipe.

More font tweaks

Once you start looking into the details of these fonts you are likely to notice more issues. For instance, in the nice looking Lucida math fonts the relations have inconsistent widths and even shapes. This can partially be corrected by using a stylistic alternate but even that forced us to come up with a mechanism to selectively replace 'bad' shapes because there is not that much granularity in the alternates. And once we looked at these alternates we noticed that the definition of of script versus calligraphic is also somewhat fuzzy and font dependent. That made for yet another tweak where we can swap alphabets and let the math machinery choose the expected shape. In Unicode this is handled by variant selectors which is rather cumbersome. Because these two styles are used mixed in the same document, a proper additional alphabet would have made more sense. As we already support variant selectors it was no big deal to combine that mechanism with a variant selector features over a range of calligraphic or script characters, which indeed is what mathematicians use (Mikael can be very convincing). With this kind of tweaks the engine doesn't really play a role: we always could and did deal with it. It's just that upgrading the engine made us look again at this.

Normalization

Once we had all these spacing related features upgraded it was time to move to other aspects math typesetting. Most of that is not handled in the engine but at the macro level. Examples of this are making sure that math spacing obeys the rules across alignment cells, breaking long formulas into lines with various alignment schemes. The first is accommodated by using the primitives that set the states at the beginning and end of a formula so that is definitely something that the engine facilitates. The second was already possible in MkIV but is somewhat more transparent now by using tagged boundary nodes.

But for this summary we stick to discussing the more low level features and where most of what we discussed here concerns horizontal spacing we also have some vertical magic like the mentioned scaled fences and operators but they sort of behave as expected given the traditional TEX approach. We have some more:

```
\definemathradical[esqrt][sqrt][height=\maxdimen,depth=\maxdimen]
\definemathradical[ssqrt][sqrt][height=3ex,depth=2ex]

\def\TestSqrt#1%
  {test $ #1{x} + #1{\sin(x)} $ test\quad
   test $ #1{x} + #1{\sin(x)} + #1{\frac{1}{x}} $ test\quad
   test $ #1{x} + #1{x^2} $ test\quad
   test $ \left(#1{x} + #1{x^2} \right) $ test\par}

\TestSqrt \sqrt  \blank
\TestSqrt \esqrt \blank
\TestSqrt \ssqrt \blank
```

test $\sqrt{x} + \sqrt{\sin(x)}$ test test $\sqrt{x} + \sqrt{\sin(x)} + \sqrt{\frac{1}{x}}$ test test $\sqrt{x} + \sqrt{x^2}$ test test $\left(\sqrt{x} + \sqrt{x^2}\right)$ test

test $\sqrt{x} + \sqrt{\sin(x)}$ test test $\sqrt{x} + \sqrt{\sin(x)} + \sqrt{\frac{1}{x}}$ test test $\sqrt{x} + \sqrt{x^2}$ test test $\left(\sqrt{x} + \sqrt{x^2}\right)$ test

test $\sqrt{x + \sqrt{\sin(x)}}$ test test $\sqrt{x + \sqrt{\sin(x)} + \sqrt{\frac{1}{x}}}$ test test $\sqrt{x + \sqrt{x^2}}$ test test $\left(\sqrt{x + \sqrt{x^2}}\right)$ test

In the above example you see that square roots can be made to adapt themselves to other such roots. For this we had to add an additional pass. Originally there are just two passes: a first typesetting pass where the maximum height and depth are collected so that in the second pass the fences can be generated and injected. That second pass also handles the spacing and penalties. In LuaMetaTEX we now have (1) radical body typesetting, (2) radical typesetting, (3) atom typesetting with height and depth analysis, (4) fence typesetting, and finally (5) inject spacing, penalties, remove slack, etc.

In the examples above we set the height and depth and these are passed by keywords to the radical primitive (most atoms and math structures accept keywords that control rendering). Here the special values `\maxdimen` signal that we have to make radicals of equal height and depth.

In MkII we had ways to snap formulas so that we got consistent line spacing. For a while I wondered if the engine could help with that but in the end no specific engine features are needed, but is is definitely an area that I keep an eye on because consistent spacing is important. After all one has to draw aline somewhere and we always have the Lua callback mechanism available.

More goodies

This summary will never be complete because we keep improving the rendering of math. For instance, when Mikael checked some less used math alphabets of Latin Modern and Bonum, as part of the goodie file completion, we were a bit horrified by the weird top accent anchoring, inconsistent dimensions and stale italic correction present in some glyphs. For instance there was a italic correction after an upright blackboard lowercase 'f', the upright digits had somewhat random top accent anchors, and due to the lack of granularity in for instance wide hats, characters that are often seen together got inconsistent wide hats. Also clashing with scripts was possible. All this resulted in yet another bunch of features:

- In the goodie files we added efficient options to remove anchors from alphabets (or individual characters).
- In the goodie files we added similar options to remove italic correction.
- Characters got a few extra fields: margins that can be used to cheat with dimensions so that we can get more consistent wide accents.
- The engine also got the possibility to compensate for accents when superscripts need to be anchored (by diminishing the height of accents as well as via an offsets).

We expect to add (and use) some more options like this when we run into other persistent issues. For sure there are some already that are not discussed here. Of course one can argue why we spend time on this: in 15 years of Unicode math usage in the TEX community no one ever bothered about a wide hat over the digit 7 and no one wondered about the bad spacing after a lowercase blackboard f, but as we go on we do run into these phenomena and it has become a bit of an obsession to get it all right.[12]

By closely looking at default positioning of accents on top of characters Mikael noticed that the anchor points are actually always in the middle of the topmost left and right points of shapes. It looks like these points are calculates automatically and therefore you end up with an anchor on top of the highest part of the seven in Latin Modern Serif but in the middle of a seven with a flat top. You also get anchors at the top of the vertical line in b, d, and on the sticky bit of the g. It is a good example of being careful with automating font design. In our case, removing most anchors and adding a few later on was the solution.[13]

Untold stories

There are of course more features but not all make sense to discuss here. For instance, all hard coded properties are now configurable. Take for instance:

```
\Umathsuperscriptvariant\textstyle  1
\Umathsubscriptvariant  \textstyle  1

$ 1_2^3 \quad {\scriptstyle 1_2^3} \quad {\displaystyle 1_2^3}$
```

This gives us:

$$1_2^3 \quad {\scriptstyle 1_2^3} \quad {\displaystyle 1_2^3}$$

Here the number refers to one of the build in variants, that themselves are a range of styles. In the next table the narrow variants are cramped:

0	**normal**	D	D	T	T	S	S	SS	SS
1	**cramped**	D	D	T	T	S	S	SS	SS
2	**subscript**	S	S	S	S	SS	SS	SS	SS
3	**superscript**	S	S	S	S	SS	SS	SS	SS

[12] Of course all this puts the usual bashing of Microsoft Word by users in a different perspective: limited control in the TEX engine, faulty fonts that come with TEX distributions, lack of testing and quality control, and probably the believe that all gets done well automatically plays a role here.

[13] In the original fonts and traditional TEX engine a kerning pair between a so called skew char and the character at hand is used.

4	**small**	S	S	S	S	SS	SS	SS	SS
5	**smaller**	S	s	S	s	SS	ss	SS	ss
6	**numerator**	S	s	S	s	SS	ss	SS	ss
7	**denominator**	s	s	s	s	ss	ss	ss	ss
8	**double (superscript)**	S	s	S	s	SS	ss	SS	ss

If you want you can change these values but of course we're then basically changing some of logic behind math rendering and for sure Don Knuth had good reasons for these defaults.

Another untold story relates to multi scripts. When a double script is seen, TEX injects an ordinary recovery atom, issues an error message, and when told so just continues. In order to always continue LuaMetaTEX introduces a mode variable that default to minus one, as negative values will trigger the error. Zero of positive values are interpreted as a class and bypass the error. Here is an example of usage:

```
\mathdoublescriptmode
  "\tohexadecimal\mathexperimentalcode % experimental class
   \tohexadecimal\mathexperimentalcode % we have to set both left and right

\setmathspacing \mathexperimentalcode \mathexperimentalcode \allmathstyles 20mu
\setmathspacing \mathordinarycode     \mathexperimentalcode \allmathstyles 20mu

$x^1_2^3_4^^^5___6^^^7___8$
```

We get this:

$$x^{1\quad 53\quad 7}_{2\quad 64\quad 8}$$

Final words

One can argue that all these new features can make a document look better. But you only have to look at what Don Knuth produces himself to see that one always could do a good job with TEX, although maybe at the cost of some extra spacing directives. It is the fact that OpenType showed up as well as many more math fonts, all with their own (sometimes surprising) special effects, that made us adapt the engine. Of course there are also new possibilities that permit better and more robust macro support. The TEXbook has a chapter on "the fine points of mathematics typesetting" for a reason.

There has never been an excuse to produce bad looking documents. It is all about care. For sure there is a category of users who are forced to use TEX, so they are excused. There are also those who have no eye for typography and rely on the macro package, so there we can to some extent blame the authors of those packages. And there are of course the sloppy users, those who don't enter a revision loop at all. They could as well use any system that in some way can handle math. One can also wonder in what way massive remote editing as well as collaborative working on documents make things better. It probably becomes less personal. At meetings and platforms TEX users like to bash the alternatives but in the end they are part of the same landscape and when it comes to math they dominate. Maybe there is less to brag about then we like: just do your thing and try to do it as good as possible. Rely on your eyes and pay attention to the details, which is possible because the engine provided the means. The previous text shows a few things to pay attention to.

Now that all the basics that have to do with proper dimensions, spacing, penalties and logic are dealt with, we moved on to the more high level constructs. We also haven't applied some features in the ConTEXt code

base yet and are now experimenting with the more high level constructs. For instance the frequently used math alignment mechanism has been overhauled to support advanced inter atom spacing across rows, and in practice one will now more often not even use this alignment mechanism and use the alignment features in multi-line display math, if only because they offer advanced annotation. As a nice side effect some of the mechanism that we use for this (like the improved `\vadjust` primitive engine feature) also became somewhat more powerful in regular text mode and we'll see where that brings us.

Given the time we spend on this and given the numerous new features it will take a while before all that got added to the engine will be documented. Of course the usage in ConTEXt also serves as documentation. This is not really a problem because most users will happily rely on the goodie files being okay and maintained, and usage other than ConTEXt is unlikely to use these new features, if only because it will break away from the established long term standards and habits.

6 The binary

This is a very short chapter. Because LuaMetaTEX is also a script runner, I want to keep it lean and mean. So, when the size exceeded 3MB after we'd extended the math engine, I decided to (finally) let all the MetaPost number interfaces pass pointers which brought down the binary 100K and below the 3MB mark again.

I then became curious about how much of the binary actually is taken by MetaPost, and a bit of calculation indicated that we went from 20.1% down to 18.3%. Here is the state per May 13, 2022:

component	pct	bytes	comment
liblua	11.8	349158	lua core, tex interfaces
libluaoptional	2.4	70263	framework, several small interfaces, cerf
libluarest	1.9	55911	general helper libraries
libluasocket	2.4	71640	helper that interfaces to the os libraries
libmimalloc	4.1	121186	memory management partial
libminiz	1.2	34962	minimalistic core
libmp	18.3	540615	mp graphic core, number libraries, lua interfacing
libpplib	7.4	220386	pdf reading core, encryption helpers
libtex	50.5	1495970	extended tex core
luametatex		**2960091**	2022-05-13

It is clear that the TEX core is good for half of the code (50.5%) with the accumulated Lua stuff (18.5%) and MetaPost being a good second (18.3%) and third and the pdf interpreting library a decent fourth (7.4%) place.

7 To the point

In the 2022 ntg Maps 53 there is a visual very attractive article about generative graphics with MetaPost by Fabrice Larribe. These graphics actually use very little MetaPost code that use randomized paths and points and the magic is in getting the parameters right. This means that one has to process them a lot to figure out what looks best. Here is an example of such a graphic definition. I will show more variants so a rendering happens later on.

```
\startMPdefinitions
    vardef agitate_a(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := for i=1 upto nbpoints:
                point (i/nbpoints) along R
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

I will not explain the working of this because there is the article. Instead, I will focus on something that came up when the Maps was prepared: performance. Not only are these graphics large (which is no real problem) but they also take a while to render (which is something that does matter when one wants to find the best a parameters). For the first few variants we keep the same names of variables as in the article.

In figure 7.1 we show the (kind of) graphic that we are dealing with. Such an agitator is used in a loop so that we agitate multiple circles, where we go from large to small with for instance 4868, 4539, 4221, 3892, 3564, 3245, 2917, 2599, 2270, 1941, 1623, 1294, 966, 647 and 319 points. The article uses a definition like below for the graphic where you can see the agitator being applied to each of the circles.

```
path P ; numeric NbCircles, S, nzero, fn, tzero, ft ;

randomseed   := 10 ;
defaultscale := .05 ;

NbCircles := 15 ; S := 10 ; nzero := 10 ; fn := 1.3 ; tzero := 5 ; ft := 0.8 ;

for c = NbCircles downto 1 :
    P := fullcircle scaled (c*6.5) scaled 3 ;
    P := agitate_a(P, S, nzero, fn, tzero, ft) ;
    eofill P
        withcolor transparent(1,4/NbCircles,col) ;
    draw P
        withpen pencircle scaled 0.1
```

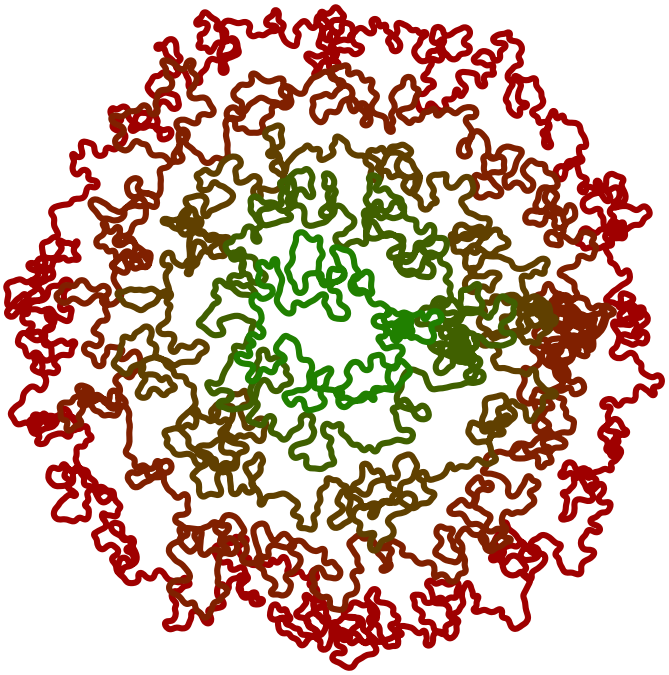


Figure 7.1 Fabrice's agitated circles, with reduced properties to keep this file small (see source).

```
        transparent(1,4/NbCircles,.90[black,col]) ;
endfor ;
```

The first we noticed is that the graphics processes faster when double mode is used: we gain 40–50% and the reason for this is that modern processors are very good at handling doubles while MetaPost in scaled mode has to do a lot of juggling with pseudo fractions. In the timings shown later we leave that improvement out. Also, because of this observation ConTEXt LMTX now defaults its MetaPost instances to method double.

When I stared at the agitator code I noticed that the `along` macro was used. That macro returns a point at given percentage along a path. In order to do that the macro calculates the length of the path and then locates that point. The primitive operations involved are `arclength`, `arctime of` and `point of` and each these takes some time to complete. A first improvement is to inline the `along` and hoist the length calculation outside the loop.

```
\startMPdefinitions
    vardef agitate_b(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (point (arctime (i * rlength) of R) of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
```

```
    enddef ;
\stopMPdefinitions
```

There is not that much that we can improve here but because Mikael Sundqvist and I had just extended MetaPost with some intersection improvements, it made sense to see what we could do in the engine. In the next variant the `arcpoint` combines `arctime of` and `point of`. The reason this is much faster is that we are already on the right spot when we got the time, and we save a sequential `point of` lookup, something that takes more time when paths are longer.

```
\startMPdefinitions
    vardef agitate_c(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (arcpoint (i * rlength) of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

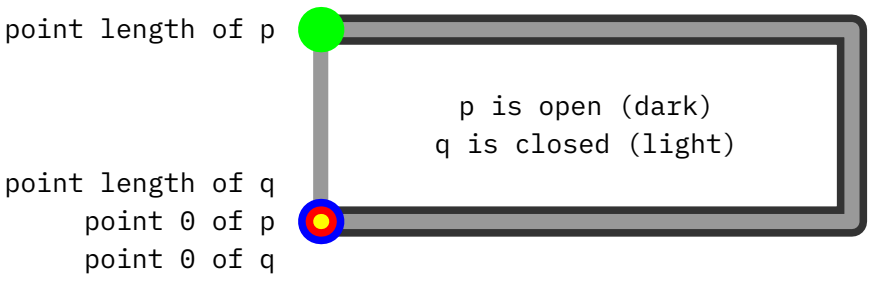
At that stage we wondered if we could come up with a primitive like `intersectiontimelist` for these points; here a list refers to a path in which we collect the points. Now, as with the intersection primitives, MetaPost loops over the segments of a path and works within such a segment. That is why the following variant has an explicit start at point zero: we can now use offsets (discrete points).

```
\startMPdefinitions
    vardef agitate_d(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (arcpoint (0, i * rlength) of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

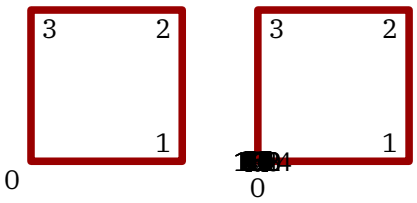
During an evening zooming Mikael and I figured out, by closely looking at the source, how the arc functions work and how we could indeed come up with a list primitive. The main issue was to use the right information. Mikael sat down to make a pure MetaPost variant and I hacked the engine. Mikael came up with a first variant similar to the following, where we use a new primitive subarclength.

```
\startMPdefinitions
    vardef arcpoints_a(expr thepath, cnt) =
        save len, seg, tot, tim, stp, acc ;
        numeric len ; len := length thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        %
        numeric acc[] ; acc[0] := 0 ;
        for i = 1 upto len:
            acc[i] := acc[i-1] + subarclength (i-1,i) of thepath ;
        endfor;
        %
        numeric stp ; stp := acc[len] / cnt;
        %
        point 0 of thepath
        for tot = stp step stp until acc[len] :
            hide(
                forever :
                    exitif ((tim < tot) and (tot < acc[seg+1])) ;
                    seg := seg + 1 ;
                    tim := acc[seg] ;
                endfor ;
            )
            -- (arcpoint (seg,tot-tim) of thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

Getting points of a path is somewhat complicated by the fact that the length of a closed path is different from that of an open path even if they have the same number of so-called knots. Internally a path is always a closed loop. That way, when MetaPost runs over a path, it can easily access the first point when it's at the end, something that is handy when that point has to be taken into account. Therefore, the end condition of a loop over a path is the arrival at the beginning. In the next graphic we show a bit how these first (zero) and last points are located. One reason why the previous macros start at point one and not at zero is that arclength can overflow due to the randomly growing path otherwise.



The difference between starting at zero or one for a cycle is show below, we get more and more points!



In the next variants we will not loop over points but step to the `arclength`. Watch the new `subarclength` primitive that starts at an offset. This is much faster than taking a `subpath of`. We can move the accumulator loop into the main loop:

```
\startMPdefinitions
    vardef arcpoints_b(expr thepath, cnt) =
        save len, aln, seg, tot, tim, stp, acc ;
        numeric len ; len := length thepath ;
        numeric aln ; aln := arclength thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        numeric stp ; stp := aln / cnt;
        numeric acc ; acc := subarclength (0,1) of thepath ;
        %
        point 0 of thepath
        for tot = stp step stp until aln :
            hide(
                forever :
                    exitif tot < acc ;
                    seg := seg + 1 ;
                    tim := acc ;
                    acc := acc + subarclength (seg,seg+1) of thepath ;
                endfor ;
            )
            -- (arcpoint (seg,tot-tim) of thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

If you don't like the `hide` the next variant also works okay:

```
\startMPdefinitions
    vardef mfun_arc_point(text tot)(text thepath) =
        forever :
            exitif tot < acc ;
            seg := seg + 1 ;
            tim := acc ;
            acc := acc + subarclength (seg,seg+1) of thepath ;
        endfor ;
        (arcpoint (seg,tot-tim) of thepath)
    enddef ;

    vardef arcpoints_c(expr thepath, cnt) =
        save len, aln, seg, tot, tim, stp, acc ;
```

```
        numeric len ; len := length thepath ;
        numeric aln ; aln := arclength thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        numeric stp ; stp := aln / cnt;
        numeric acc ; acc := subarclength (0,1) of thepath ;
        %
        point 0 of thepath
        for tot = stp step stp until aln :
            -- mfun_arc_point(tot)(thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

This got applied in three test agitators

```
\startMPdefinitions
    vardef agitate_e_a(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_a(R, nbpoints) ; % original Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;

    vardef agitate_e_b(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_b(R, nbpoints) ; % merged Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
```

```
    enddef ;

    vardef agitate_e_c(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_c(R, nbpoints) ; % split Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

The new engine primitive shortens these agitators:

```
\startMPdefinitions
    vardef agitate_e_d(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpointlist nbpoints of R;
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

So are we done? Did we get rid of all bottlenecks? The answer is no! We still loop over the list in order to randomize the points. For each point we start at the beginning of the list. Let's first rewrite the agitator a little:

```
\startMPdefinitions
    vardef agitate_f_a(expr pth, iterations, points, pointfactor, noise,
                                                                noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
```

```
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentpath   := arcpointlist currentpoints of currentpath ;
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            if currentnoise <> 0 :
                currentpath :=
                    for i = 0 upto length currentpath:
                        (point i of currentpath) randomized currentnoise ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

One of the LuaMetaFun extensions is a fast path iterator. In the next variant the `inpath` macro sets up an iterator (regular loop) with the length as final value. In the process the given path gets passed to Lua where we can access it as array. The `pointof` macro (again a Lua call) injects a pair. You will be surprised that even with passing the path to Lua and calling out to Lua to inject the pair this is way faster than the built-in `point of`.

```
\startMPdefinitions
    vardef agitate_f_b(expr pth, iterations, points, pointfactor, noise,
                                                                    noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            currentpath   := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
                currentpath :=
                    for i inpath currentpath :
                        (pointof i) randomized currentnoise ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

It was tempting to see if a more native solution pays of. One problem there is that a path is not really suitable for that as we currently don't have a data type that represents a point. Okay, actually we sort of have because we can use the transform record that has six points but that is something I will look into later (it just got added to the todo list).

The `i within pth` iterator is no conceptual beauty but does the job. Just keep in mind that it is just means for this kind of applications: run over a path point by point. The `i` has the current point number. Because we run over a path following the links we only run forward.

```
\startMPdefinitions
    vardef agitate_f_c(expr pth, iterations, points, pointfactor, noise,
                                                                  noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            currentpath   := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
                currentpath :=
                    for i within currentpath :
                        pathpoint
                        randomized currentnoise
                        ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

Any primitive solution more complex than this, like first creating a fast access data structure, of having a double linked list, or using some iterator larger than a simple numeric is very likely to have no gain over the super fast Lua variant.

We show the average runtime for three runs. Here we don't render the paths, which takes about one second, including conversion to pdf. Of course measurements like this can change a bit over time. To these times you need to add about a second for the draw and fill operations as well as conversion to a pdf stream with transparencies. The improvement in runtime makes it possible to use agitators like this at runtime especially because normally one will not use such (combinations of) large paths.

agitate_a	776.26	agitate_e_a	291.99	agitate_f_a	10.82
agitate_b	276.43	agitate_e_b	76.06	agitate_f_b	2.55
agitate_c	259.89	agitate_e_c	77.27	agitate_f_c	2.17
agitate_d	260.41	agitate_e_d	18.67		

The final version of the agitator is slightly different because it depends if we start at zero or one but gives similar results and adapt the noise before or after the loop.

```
\startMPdefinitions
    vardef agitator(expr pth, iterations, points, pointfactor, noise, noisefactor)
                                                                              =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
```

```
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentpath := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
                currentpath :=
                    for i within currentpath :
                        pathpoint
                        randomized currentnoise
                        ..
                    endfor
                cycle ;
            fi
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

We use a similar example as in the mentioned article but coded a bit differently:

```
\startMPcode
  path pth ;
  nofcircles  := 15 ; iterations  := 10 ;
  points      := 10 ; pointfactor := 1.3 ;
  noise       :=  5 ; noisefactor := 0.8 ;

  nofcircles  :=  5 ; iterations  := 10 ;
  points      :=  5 ; pointfactor := 1.3 ;

% for c = nofcircles downto 1 :
%   pth    := fullcircle scaled (c * 6.5) scaled 3 ;
%   points := floor(arclength(pth) * 0.5) ;
%   pth    := agitator(pth, iterations, points, pointfactor, noise, noisefactor) ;
%   eofill pth
%     withcolor darkred
%     withtransparency(1,4/nofcircles) ;
%   draw pth
%     withpen pencircle scaled 0.1
%     withtransparency(1,4/nofcircles) ;
% endfor ;

% currentpicture := currentpicture xsized TextWidth ;

  for c = nofcircles downto 1 :
    pth    := fullcircle scaled (c * 6.5) scaled 3 ;
    points := floor(arclength(pth) * 0.5) ;
    pth    := agitator(pth, iterations, points, pointfactor, noise, noisefactor) ;
    draw pth
```

```
      withpen pencircle scaled 1
      withcolor (c/nofcircles)[darkgreen,darkred] ;
  endfor ;

  currentpicture := currentpicture xsized .5TextWidth ;
\stopMPcode
```

For Mikael and me, who both like MetaPost, it was a nice distraction from working months on extending math in LuaMetaTEX, but it also opens up the possibilities to do more with rendering (math) functions and graphics, so in the end we get paid back anyway.

8 Not all makes sense

The development of ConTEXt is to a large extend driven by users with a wide variety of background and usage. I can safely say that much time spent on ConTEXt qualifies as hobby (or maybe even more by curiosity). Of course I do use it myself but personally I never make advanced documents. I'm not a writer, nor an artist, nor a typesetter. I do like challenges so that's why we get mechanisms that can do tricky things and some stay sort of hidden because the practical usage is limited, although you will be surprised to see what users find in the source and use anyway. My colleague uses ConTEXt for large scale, mostly complex and demanding xml documents where one source is rendered in different ways with different parts used. Many features in ConTEXt relate to workflows.

I like to visualize things so that's part of the development cycle. I never start from some 'typographical' point of view, if only because in my experience much design is arbitrary and personal. The output should look okay on the average, and on reasonable simple documents there should be no need for manual intervention. I am quite willing to accept an occasional less optimal looking page and don't loose sleep over it. A next time, when a sentence gets added, it might be better and the problem can be moved further down the pages. Also, given what one runs into nowadays the average job that TEX does is pretty good (but users can of course mess up). It is boundary conditions that determine in what direction a style or solution goes. The more abstract one argues about typesetting and possible solutions, the less interested I often become simply because there are no perfect solutions for every case. There are always those last few % points that need manual intervention or some trickery and most users get that. It is also what makes using TEX fun.

As mentioned, the TEX engine does a pretty good job on average but that didn't prevent me from extending it: the mix of TEX, MetaPost and Lua is even more fun. But what is the development agenda there? Again, it is very much driven by what users want me to solve, but there's also the curiosity element. A recent example of extending is the math sub system. It was already made more configurable and some features where added but now it is really flexible. This was doable because the heuristics in the engine are clear. It was could be done because I had a dedicated partner in this journey.[14] Other parts are more difficult but have nevertheless been extended, to mention a few: alignments, par building and page building. However the last two use some heuristics that are hard to make more flexible. For instance the badness calculation combined with the loop that tries to find breakpoints is already quite good and the somewhat special values involved in the calculations have been optimized stepwise by Don Knuth during the development of TEX.

Does that mean that one cannot add some options to influence that tuning? For sure one can. The source has this comment:

> "When looking for optimal line breaks, TEX creates a 'break node' for each break that is *feasible*, in the sense that there is a way to end a line at the given place without requiring any line to stretch more than a given tolerance. A break node is characterized by three things: the position of the break (which is a pointer to a `glue_node`, `math_node`, `penalty_node`, or `disc_node`); the ordinal number of the line that will follow this breakpoint; and the fitness classification of the line that has just ended, i.e., `tight_fit`, `decent_fit`, `loose_fit`, or `very_loose_fit`."

The book TEX by Topic (by Eijkhout) gives a good explanation of the way lines are broken so there is no need to go into detail here. The code involved is not that trivial anyway. The criteria for deciding what is bad are as follows:

verdict	effect	badness

[14] In another chapter I summarize what Mikael Sundqvist and I did in this context.

very loose	stretch	>= 100
loose	stretch	>= 13
decent		<= 12
tight	shrink	>= 13

When the difference between two lines is more than one, they are considered to be visually incompatible. Then, if the badness of any line exceeds `pretolerance` a second pass is triggered, When `pretolerance` is negative the first pass is skipped. When the badness of any line exceeds `tolerance` a third pass is triggered and `emergencystretch` is used to make things fit.

Where in traditional TEX a lot of parsing, hyphenation, font handling and par building is combined, in LuaMetaTEX we always work with completely hyphenated and font readied lists. In traditional TEX the first pass works on the original non-hyphenated lists.

In the source there is an old note that one day I will play with a plugged in badness calculation but it also says that there might be a performance impact as well as all kind of unforeseen side effects because TEX makes sure that the heuristics lead to values that don't result in overflow and such.

Another note concerns more fitness values. Doing that will increase the runtime a little but on a modern machine that is not really an issue. Shortly after I upgraded my laptop to a somewhat newer one I decided to play with this and therefore any performance hit would go unnoticed anyway. The following snippet from the source shows the idea:

```
typedef enum fitness_value {
    very_loose_fit, /*tex lines stretching more than their stretchability */
    loose_fit,      /*tex lines stretching 0.5 to 1.0 of their stretchability */
    semi_loose_fit,
    decent_fit,     /*tex for all other lines */
    semi_tight_fit,
    tight_fit,      /*tex lines shrinking 0.5 to 1.0 of their shrinkability */
    n_of_finess_values
} fitness_value;
```

This means that when we loop over `very_loose_fit` upto `tight_fit` we have two more classes to take into account: the semi ones. Playing with that and associating them with magic numbers quickly learned that we enter the area of 'random improvements'. You can render variants and because some will look better and others worse one can argue for any case. And as usual, once a user (unaware of what we are doing) looks at it, things like successive hyphens, wider spaces, rivers and such are seen as the main difference. Of course spacing is the direct result of this kind of messing, but because the effects are actually mostly noticeable on non-justified texts it then is the end-of-line spacing that influences the verdict.[15]

In the end this kind of extensions make little sense. One can of course play science and introduce all kind of imaginary cases where it might work but that is why I started this summary by explaining what drives developments: users and constraints. Playing science for the sake of it is pseudo science. And, as with much science related to typesetting (probably with the exception of Don's work) most has therefore little practical value.

[15] When hz showed up in pdfTEX we did experiments with random samples of its usage and TEXies at user group meetings and the results were such that one could only draw the conclusion that on the average a user has no clue if something is good or bad for what reason. The strong emphasis in the TEX community on hyphenation makes that an eye-catching criterium. So having two in a successive lines even when there is really no better solution is what draws the attention and users then tend to think that what a survey is about is "The quality of hyphenation related to breaking paragraphs into lines."

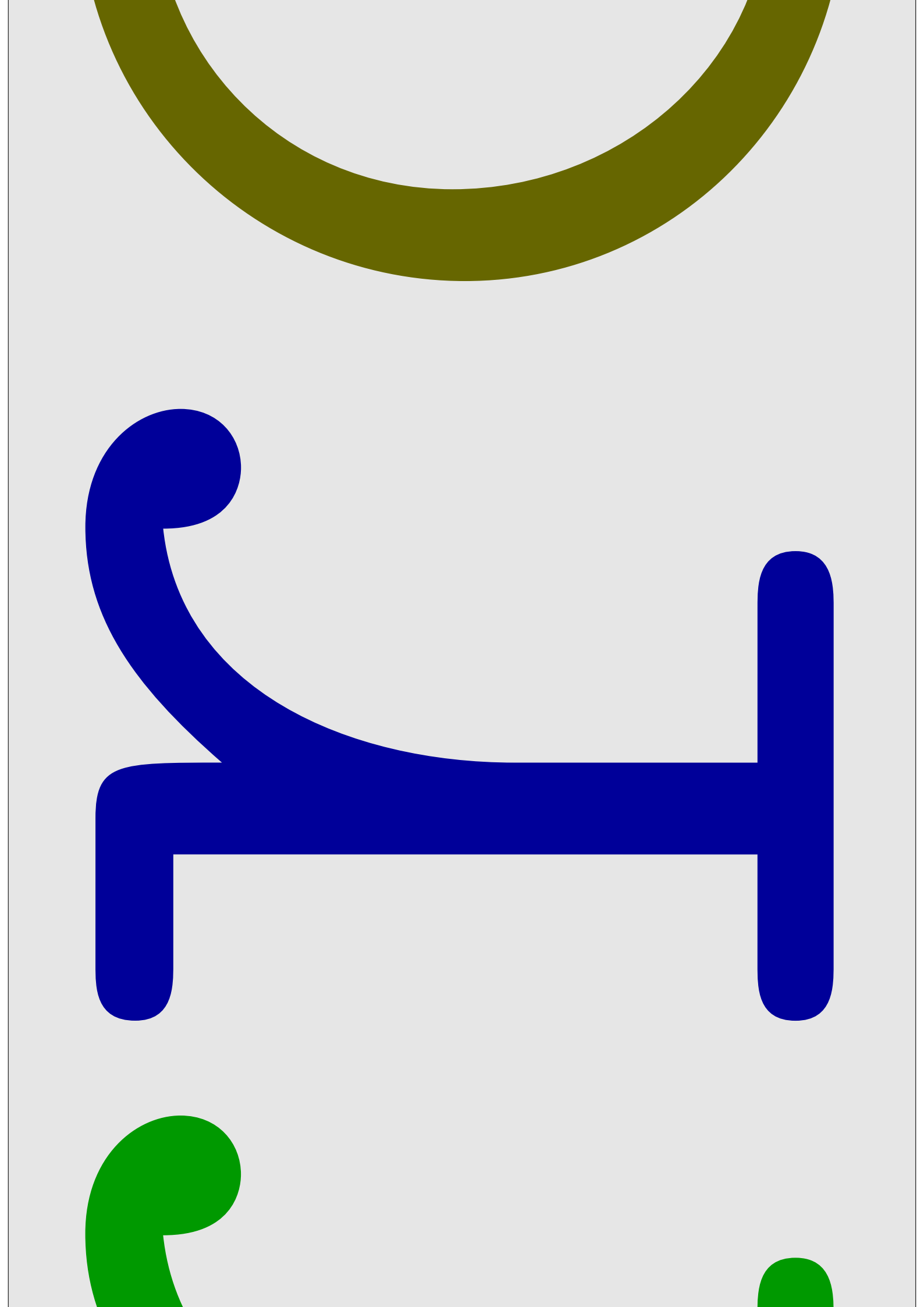
So, do we keep this feature or not? We actually do, if only to be able to demonstrate the fuzziness of this. We have an undocumented magic parameter:

```
\linebreakcriterium"0C0C0C63
```

Actually the value is zero but when one of the four byte pairs is zero it will default to `"0C` (12) or `"63` (99). The values concern `semitight`, `decent`, `semiloose`, and `loose`. After some trial and error I got to the examples on the next two pages. You need to zoom in to see the differences (the black one is the original). In setting used are:

	`\hsize`	`\setupalign`
1	12em	normal, stretch, tolerant
2	18em	flushleft

As mentioned, one can look at specific expected properties and draw conclusions but when TEX cannot find a good solution using its default, it is unlikely that alternative settings help you out, unless you do that on a per-paragraph basis.



9 But this does

In LuaMetaTEX one can do a lot on Lua, like what I will discuss next, but because it is somewhat fundamental it became a core feature of the engine. It was also quite easy to implement. It has to do with packaging.[16]

The box constructors in traditional TEX accept two keywords: `to` for setting an exact width and `spread` for specifying additional width. In LuaMetaTEX we have some more like `shift` (a traditional TEX concept), `orientation`, `xmove`, `xoffset`, `ymove` and `yoffset` for absolute positioning, `anchor(s)`, `target` and `source` for relative positioning, `axis` and `class` for usage in , `delay` for leader like boxes, the multiple `attr` key for setting attributes, a special subtype directive `container` and `direction` for controlling bidirectional typesetting, and `reverse` for reversing content. The latest addition: `adapt` is there for controlling and freezing glue.

So, in addition to the width related keys `to` and `spread` we have `adapt` that drives wo what width the box will be typeset.[17] The keyword is followed by a scale values between -1000 and 1000 where a negative value enforces shrink and a positive value stretch. The following table shows the effects:

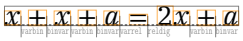
	Here are just some words so that we can see what happens.
`to 4cm`	Here are just some words so that we can see what happens.
`to \hsize`	Here are just some words so that we can see what happens.
`spread 1cm`	Here are just some words so that we can see what happens.
`spread -1cm`	Here are just some words so that we can see what happens.
`adapt -1000`	Here are just some words so that we can see what happens.
`adapt -750`	Here are just some words so that we can see what happens.
`adapt -500`	Here are just some words so that we can see what happens.
`adapt 0`	Here are just some words so that we can see what happens.
`adapt 500`	Here are just some words so that we can see what happens.
`adapt 750`	Here are just some words so that we can see what happens.
`adapt 1000`	Here are just some words so that we can see what happens.

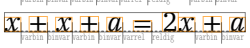
When a box is typeset the natural glue width is used but when the required width exceeds the natural width the glue stretch components kick in. With a negative spread the shrink is used but you can get underflows. The `adapt` feature freezes the glue and it removes the stretch and shrink after applying it to the glue width with the given scale factor. So, in order to get the minimum width you use `adapt -1000`.

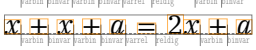
The reason why I decided to add this feature is that when experimenting with math alignments I wanted to be able to see what shrink could be achieved.[18] The next example shows this:

	$x + x + a = 2x + a$
`to 4cm`	$x + x + a = 2x + a$
`to \hsize`	$x + x + a = 2x + a$
`spread 1cm`	$x + x + a = 2x + a$
`spread -1cm`	$x + x + a = 2x + a$
`adapt -1000`	$x+x+a=2x+a$
`adapt -750`	$x+x+a=2x+a$
`adapt -500`	$x+x+a=2x+a$
`adapt 0`	$x+x+a=2x+a$

[16] I actually did prototype it in Lua first but wanted a more natural integration in the end.
[17] For the moment this keyword only has effect for horizontal boxes.
[18] At that time Mikael and I were experimenting with consistent spacing in math alignments.

adapt 500 

adapt 750 

adapt 1000 

Once we had this new feature it made sense to add support for it to `\framed`, one of the oldest macros that got extended over time:

```
\inframed[adaptive=1000] {Just some words}
\inframed[adaptive=500]  {Just some words}
\inframed[adaptive=0]    {Just some words}
\inframed[adaptive=-500] {Just some words}
\inframed[adaptive=-1000]{Just some words}
```

This renders as:

Just some words
Just some words
Just some words
Just some words
Just some words

Once we have it there other mechanisms can benefit from it, for instance natural tables. But keep in mind that spaces are fixed in there so there is only the expected result if glue has stretch or shrink.

10 The curious case of \over

Normally TEX scans forward but there are a few special cases. First of all, TEX is either scanning regular content or it is scanning alignments. That results in intercepts in all kind of places. When a row ends, scanning for inter-row actions happens. When the preamble is scanned there is some lookahead with partial expansion. This has side effects but these can be avoided in LuaMetaTEX by several options. Another special case is math mode. Normally curly braces indicate grouping but not in math mode: there they construct an atom, ordinary by default. Although most math constructs actually pick up some following atom, in which case we get a wrapped construct that actually is processed in a nested cal to the math processing routine. That whole has a class and although in LuaMetaTEX we can make atoms with a different left end right class, normally what is inside is hidden stays hidden.[19]

Fraction commands like `\over` and `\above` are used like this:

```
a + 1 \over 2 + b
```

and as there can be more than for instance single digits we can do:

```
a + {12} \over {34} + b
```

but it doesn't end there because you actually need to wrap:

```
a + {{12} \over {34}} + b
```

If you don't do this *b* will become part of the fraction. The curly braces here make the 12, 34 and the whole fraction made from them ordinary atoms. Because that also influences spacing one should be aware of side effects.

```
a + {{12} \over {34}} + b
```

The argument of `simplicity` of input is easily defeated by using

```
a + \frac{12}{34} + b
```

because it also reads sequential, is in sync with other commands like `\sqrt` and actually uses less tokens. It used more runtime, also because ConTEXt adds plenty of control and extras but you won't notice it.

Because in ConTEXt we assume users to use `\frac` it made sense to see if we can make curly braces act like groups. In LuaMetaTEX we already have `\beginmathgroup` and `\endmathgroup` that provide grouping with mathstyle recovery, and by setting `\mathgroupingmode` to a non-zero value curly braces will act like these.

The effects are subtle:

```
$ a + { \bf x } ^ 2  +  {\bf 1} + {\red 123} +
\frac{1}{2} + {\scriptstyle 123} + \dd $
```

[19] We can think of optionally exposing the edge classes but although it is easy to implement we see no reason to do that now. After all, the information is actually available already via variables.


```
\mathgroupingmode 0
```


```
\mathgroupingmode 1
```

If you see the differences you might be happy with this new trick, if not, you probably are not that much into optimal math spacing anyway and you can forget about what you just read.

11 Getting rid of jit

At the ntg meeting there was a short discussion about performance of the OpenType font machinery. Currently we still support LuajitTEX and although the MkIV code base is mostly separated from the LMTX one there is still some compromise going on, as some Lua code is shared and needs to adapt to the fact that LuaJIT is stuck to Lua5.2 (sort of). One reason why we still support LuajitTEX is that there are users who like the performance gain. However, if these can switch to ConTEXt we could get rid of LuajitTEX support. After all, LuaJIT is stalled as is ffi. Of course a plain TEX users can object to using ConTEXt but one can just use the basics and be as plain as possible.

Performance of LuaMetaTEX is quite okay and it often performs better than LuaTEX or even LuajitTEX. One border case is for instance the somewhat overdone in terms of split feature steps is the Brill font. So, I decided to do some tests on my 2017 laptop that had replaced the 2013 one (Windows 10). We use cross compiled binaries. Here is the test:

```
% \enableexperiments[fonts.compact]

\definefont[Fa][brill*default    @ 10pt]
\definefont[Fb][brill*default    @ 12pt]
\definefont[Fc][brill-bf*default @ 10pt]
\definefont[Fd][brill-bf*default @ 12pt]

\start \testfeatureonce{1000}{
    \Fa \samplefile{tufte}       \samplefile{tufte}\par
}\stop \page \edef\TimeA{\elapsedtime}
\start \testfeatureonce{1000}{
    \Fa \samplefile{tufte}\par\Fb\samplefile{tufte}\par
}\stop \page \edef\TimeB{\elapsedtime}
\start \testfeatureonce{1000}{
    \Fa \samplefile{tufte}    \Fb\samplefile{tufte}\par
}\stop \page \edef\TimeC{\elapsedtime}
\start \testfeatureonce{1000}{
    \Fa \samplefile{tufte}    \Fd\samplefile{tufte}\par
    \Fb \samplefile{tufte}    \Fc\samplefile{tufte}\par
}\stop \page \edef\TimeD{\elapsedtime}

\startTEXpage[offset=10pt]
    \strut\infofont
    2 par 1 font : \TimeA\par
    2 par 2 font : \TimeB\par
    1 par 2 font : \TimeC\par
    1 par 4 font : \TimeD\par
\stopTEXpage
```

The next table shows the best times from three tests where each one produces 1693 pages in the default layout. We're talking of one run as normally ConTEXt will run till a two pass stable state is reached (unless it is forced to one run), although in practice fixing a few typos will not for an extra run. Keep in mind that in LuaMetaTEX we have a Lua driven backend that is more flexible with respect to fonts but therefore also adds some extra overhead. Also keep in mind that four different fonts per paragraph is a rare case.

	2 pars 1 font	2 pars 2 fonts	1 par 2 fonts	1 par 4 fonts
luametatex	8.9	9.4	12.1	24.6
luametatex compact	8.9	9.3	9.3	23.9
luatex	10.0	10.3	14.5	31.2
luajittex	8.0	8.1	11.4	23.2

One should take these measurements with a grain of salt because it also depends on the system load, but it shows that there is no real need to favor a LuajitTEX setup over a LuaMetaTEX one. In the meantime the default LuaTEX binaries exceed 7 MB (and the hb variant adds quite a bit more) which LuaMetaTEX stays around 3 MB which is nice for high performance setups with thousands of (small) runs.

Just for the record, when we use Dejavu Serif we get 2767 pages and the following timings. Again, the differences between LuaMetaTEX and LuajitTEX is not that significant, especially when you realize that we're not doing anything fancy that more runtime. In practice fonts are only part of the story.

	2 pars 1 font	2 pars 2 fonts	1 par 2 fonts	1 par 4 fonts
luametatex	4.2	4.4	5.0	10.8
luametatex compact	4.2	4.5	4.5	10.5
luatex	4.9	5.0	6.2	13.3
luajittex	4.0	4.0	5.0	10.3

12 Issues in math fonts

12.1 Introduction

After trying to improve math rendering of OpenType math fonts, we[20] ended up with a mix of improving the engine and fixing fonts runtime, and we are rather satisfied with the results so far.

However, as we progress and also improve the more structural and input related features of ConTEXt, we wonder why we don't simply are more drastic when it comes to fonts. The OpenType specifications are vague, and most existing OpenType math fonts use a mixture of the OpenType features and the old TEX habits, so we are sort of on our own. The advantage of this situation is that we feel free to experiment and do as we like.

In another article we discuss our issues with Unicode math, and we have realized that good working solutions will be bound to a macro package anyway. Also, math typesetting has not evolved much after Don Knuth set the standard, even if the limitations of those times in terms of memory, processing speed and font technologies have been lifted already for a while. And right from the start Don invited users to extend and adapt TEX to one's needs.

Here we will zoom in on a few aspects: font parameters, glyph dimensions and properties and kerning of scripts and atoms. We discuss OpenType math fonts only, and start with a summary of how we tweak them. We leave a detailed engine discussion to a future article, since that would demand way more pages, and could confuse the reader.

12.2 Tweaks, also known as goodies

The easiest tweaks to describe are those that wipe features. Because the TEX Gyre fonts have many bad top accent anchors (they sit above the highest point of the shape) the `wipeanchors` tweak can remove them, and we do that per specified alphabet.

$$\hat{7}$$

In a similar fashion we `wipeitalics` from upright shapes. Okay, maybe they can play a role for subscript placement, but then they can also interfere, and they do not fit with the OpenType specification. The `wipecues` tweak zeros the dimensions of the invisible times and friends so that they don't interfere and `wipevariants` gets rid of bad variants of specified characters.

The fixers is another category, and the names indicate what gets fixed. Tweaks like these take lists of code points and specific properties to fix. We could leave it to your imagination what `fixaccents`, `fixanchors`, `fixellipses`, `fixoldschool`, `fixprimes`, `fixradicals` and `fixslashes` do, but here are some details. Inconsistencies in the dimensions of accents make them jump all over the place so we normalize them. We support horizontal stretching at the engine level.

$$\overbrace{a+b+c+d}^{\text{varbin binvar varbin binvar varbin binvar accrel}} = \overbrace{u+v+w+x+y}^{\text{relacc varbin binvar varbin binvar varbin binvar varbin binvar}}$$

It required only a few lines of code thanks to already present scaling features.

[20] Mikael Sundqvist and Hans Hagen

Anchors can be off so we fix these in a way so that they look better on especially italic shapes. We make sure that the automated sizing works consistently, as this is driven by width and overshoot. Several kind of ellipses can be inconsistent with each other as well as with periods (shape and size wise) so we have to deal with that. Radicals and other extensibles have old school dimensions (TEX fonts have a limited set of widths and heights). We need to fix for instance fences of various size because we want to apply kerns to scripts on the four possible corners for which we need to know the real height and depth,

Discussing primes would take many paragraphs so we stick to mentioning that they are a mess. We now have native prime support in the engine as well as assume properly dimensioned symbols to be used. Slashes are used for skewed fractions so we'd better make sure they are set up right.

A nice tweak is `replacealphabets`. We use this to provide alternative script (roundhand) and calligraphic (chancery) alphabets (yes we have both natively in ConTEXt while Unicode combines them in one alphabet). Many available OpenType math fonts come with one of the two alphabets only, some with roundhand and some with chancery. For the record: this tweak replaces the older `variants` tweak that filtered scripts from a stylistic font feature.

We also use the `replacealphabets` tweak to drop in Arabic shapes so that we can do bidirectional math. In practice that doesn't really boil down to a replacement but more to an addition. The `addmirrors` features accompanies this, and it is again a rather small extension to the engine to make sure we can do this efficiently: when a character is looked up we check a mirror variant when we are in r2l mode, just like we look up a smaller variant when we're in compact font mode (a ConTEXt feature).

$$(?\neq ?)\quad \frac{?+?-??-?}{?-?}??=??\sum_{?=?}^{?}$$

Another application of `replacealphabets` is to drop in single characters from another font. We use this for instance to replace the 'not really an alpha' in Bonum by one of our own liking. Below we show a math italic a and the original alpha, together with the modified alpha.

$$a+\alpha$$

For that we ship a companion font. On our disks (and in the distribution) you can find:

```
/tex/texmf-fonts/fonts/data/cms/companion/RalphSmithsFormalScript-Companion.otf
/tex/texmf-fonts/fonts/data/cms/companion/TeXGyreBonumMath-Companion.otf
/tex/texmf-fonts/fonts/data/cms/companion/XITSMath-Companion.otf
```

All these are efficient drop-ins that are injected by the `replacealphabets`, some under user control, some always. We tried to limit the overhead and actually bidirectional math could be simplified which also had the benefit that when one does tens of thousands of bodyfont switches a bit of runtime is gained.

There are more addition tweaks: `addactuarian` creates the relevant symbols which is actually a right sided radical (the engine has support for two-sided radicals). It takes a bit of juggling with virtual glyphs and extensible recipes, but the results are rewarding.

$$\overline{A}_{m}^{2}{}_{x}^{1}{}_{n}$$

In a similar fashion we try to add missing extensible arrows with `addarrows`, bars with `addbars`, equals with `addequals` and again using the radical mechanism fourier notation symbols (like hats) with `addfourier`. That one involves subtle kerning because these symbols end up at the right top of a fence like symbol.

$$\widehat{f * g * h}(\xi) = \left(f * g * h\right)\hat{}(\xi)$$

It was actually one of the reasons to introduce a more advanced kerning mechanism in the engine, which is not entirely trivial because one has to carry around more information, since all this is font and character bound, and when wrapped in boxes that gets hard to analyze. The `addrules` makes sure that we can do bars over and under constructs properly. The `addparts` is there to add extensible recipes to characters.

Some of these tweaks actually are not new and are also available in MkIV but more as features (optionally driven by the goodie file). An example is `addscripts` that is there for specially positioned and scaled signs (high minus and such) but that tweak will probably be redone as part of the "deal with all these plus and minus issues". The dedicated to Alan Braslau `addprivates` tweak is an example of this: we add specific variants for unary minus and plus that users can enable on demand, which in turn of course gives class specific spacing, but we promised not to discuss those engine features here.

$$\int_1^2 \left[(x+2)^{\frac{1}{2}} - (x+2)^{?\frac{1}{2}}\right] dx$$

There is a handful of tweaks that deals with fixing glyph properties (in detail). We mention: `dimensions` and `accentdimensions` that can reposition in the boundingbox, fix the width and italic correction, squeeze and expand etc. The `kernpairs` tweak adds kern pairs to combinations of characters. The `kerns` provides a way to add top left, bottom left, top right and bottom right kerns and those really make the results look better so we love it!

$$\left(\frac{1}{1+x^2}\right)^n \qquad x^2/(1+x)$$

The `margins` tweak sets margin fields that the engine can use to calculate accents over the base character better. The same is true for `setovershoots` that can make accents lean over a bit. The `staircase` feature can be used to add the somewhat complicated OpenType kerns. From all this you can deduce that the engine has all types of kerning that OpenType requires, and more.

Accents as specified in fonts can be a pain to deal with so we have more tweaks for them: `copyaccents` moves them to the right slots and `extendaccents` makes sure that we can extend them. Not all font makers have the same ideas about where these symbols should sit and what their dimensions should be.

The `checkspacing` tweak fixes bad or missing spacing related to Unicode character entries in the font, because after all, we might need them. We need to keep for instance MathML in mind, which means: processing content that we don't see and that can contain whatever an editor puts in. The `replacements` feature replaces one character by another from the same font. The `substitutes` replaces a character by one from a stylistic feature.

Relatively late we added the `setoptions` which was needed to control the engine for specific fonts. The rendering is controlled by a bunch of options (think of kerning, italic correction, and such). Some are per font, many per class. Because we can (and do) use mixed math fonts in a document, we might need to adapt the engine level options per font, and that is what this tweak does: it passes options to the font so that the engine can consult them and prefer them over the 'global' ones. We needed this for some fonts that have old school dimensions for extensibles (like Lucida), simply because they imitated Computer Modern. Normally that goes unnoticed, but, as mentioned before, it interferes with our optional kerning. The `fixoldschool`

tweak sort of can fix that too so `setoptions` is seldom needed. Luckily, some font providers are willing to fix their fonts!

We set and configure all these tweaks in a so-called goodie file, basically a runtime module that returns a Lua table with specifications. In addition to the tweaks subtable in the math namespace, there is a subtable that overloads the font parameters: the ones that OpenType specifies, but also new ones that we added. In the next section we elaborate more on these font bound parameters.

12.3 Font parameters

At some point in the upgrading of the math machinery we discussed some of the inconsistencies between the math constants of the XITS and STIX fonts. Now, one has to keep in mind that XITS was based on a first release of STIX that only had Type1 fonts so what follows should not to be seen as criticism, but more as observations and reason for discussion, as well as a basis for decisions to be made.

One thing we have to mention in advance, is that we often wonder why some weird and/or confusing stuff in math fonts go unnoticed. We have some suggestions:

- The user doesn't care that much how math comes out. This can easily be observed when you run into documents on the internet or posts on forums. And publishers don't always seem to care either. Consistency with old documents sometimes seems to be more important than quality.
- The user switches to another math font when the current one doesn't handle its intended math domain well. We have seen that happening and it's the easiest way out when you have not much control anyway (for instance when using online tools).
- The user eventually adds some skips and kerns to get things right, because after all TEX is also about tweaking.
- The user doesn't typeset that complex math. It's mostly inline math with an occasional alignment (also in text style) and very few multi-level display math (with left and right fences that span at most a fraction).

We do not claim to be perfect, but we care for details, so let's go on. The next table shows the math constants as they can be found in the Stix (two) and Xits (one) fonts. When you typeset with these fonts you will notice that Xits is somewhat smaller, so two additional columns show the values compensated for the axis height and accent base height.

constant	stix	xits	base	axis	relevance
AccentBaseHeight	480	450	**480**	464	optional**
AxisHeight	258	250	267	**258**	mandate
DelimitedSubFormulaMinHeight	1325	1500	1600	1548	
DisplayOperatorMinHeight	1800	1450	1547	1496	
FlattenedAccentBaseHeight	656	662	706	683	optional**
FractionDenominatorDisplayStyleGapMin	150	198	211	204	
FractionDenominatorDisplayStyleShiftDown	640	700	747	722	
FractionDenominatorGapMin	68	66	70	**68**	
FractionDenominatorShiftDown	585	480	512	495	
FractionNumeratorDisplayStyleGapMin	150	198	211	204	
FractionNumeratorDisplayStyleShiftUp	640	580	619	599	
FractionNumeratorGapMin	68	66	70	**68**	
FractionNumeratorShiftUp	585	480	512	495	

FractionRuleThickness	68	66	70	**68**	optional
LowerLimitBaselineDropMin	670	600	640	619	
LowerLimitGapMin	135	150	160	155	
MathLeading	150	150	160	155	
MinConnectorOverlap	100	50	53	52	mandate
OverbarExtraAscender	68	66	70	**68**	
OverbarRuleThickness	68	66	70	**68**	optional*
OverbarVerticalGap	175	198	211	204	
RadicalDegreeBottomRaisePercent	55	70	75	72	mandate
RadicalDisplayStyleVerticalGap	170	186	198	192	
RadicalExtraAscender	78	66	70	68	
RadicalKernAfterDegree	-335	-555	-592	-573	
RadicalKernBeforeDegree	65	277	295	286	
RadicalRuleThickness	68	66	70	**68**	
RadicalVerticalGap	85	82	87	**85**	
ScriptPercentScaleDown	70	75	80	77	
ScriptScriptPercentScaleDown	55	60	64	62	
SkewedFractionHorizontalGap	350	300	320	310	
SkewedFractionVerticalGap	68	66	70	**68**	
SpaceAfterScript	40	41	44	42	
StackBottomDisplayStyleShiftDown	690	900	960	929	
StackBottomShiftDown	385	800	853	826	
StackDisplayStyleGapMin	300	462	493	477	
StackGapMin	150	198	211	204	
StackTopDisplayStyleShiftUp	780	580	619	599	
StackTopShiftUp	470	480	512	495	
StretchStackBottomShiftDown	590	600	640	619	
StretchStackGapAboveMin	68	150	160	155	
StretchStackGapBelowMin	68	150	160	155	
StretchStackTopShiftUp	800	300	320	310	
SubSuperscriptGapMin	150	264	282	272	
SubscriptBaselineDropMin	160	50	53	52	
SubscriptShiftDown	210	250	267	258	
SubscriptTopMax	368	400	427	413	
SuperscriptBaselineDropMax	230	375	400	387	
SuperscriptBottomMaxWithSubscript	380	400	427	413	
SuperscriptBottomMin	120	125	133	129	
SuperscriptShiftUp	360	400	427	413	
SuperscriptShiftUpCramped	252	275	293	284	
UnderbarExtraDescender	68	66	70	**68**	
UnderbarRuleThickness	68	66	70	**68**	optional*
UnderbarVerticalGap	175	198	211	204	
UpperLimitBaselineRiseMin	300	300	320	310	
UpperLimitGapMin	135	150	160	155	

Very few values are the same. So, what exactly do these constants tell us? You can even wonder why they are there at all. Just think of this: we want to typeset math, and we have an engine that we can control. We know how we want it to look. So, what do these constants actually contribute? Plenty relates to the height and depth of the nucleus and/or the axis. The fact that we have to fix some in the goodie files, and the fact that we actually need more variables that control positioning, makes for a good argument to just ignore most of

the ones provided by the font, especially when they seem somewhat arbitrarily. Can it be that font designers are just gambling a bit, looking at another font, and starting from there?

The relationship between TEX's math font parameters and the OpenType math constants is not one-to-one. Mapping them onto each other is possible but actually is font dependent. However, we can assume that the values of Computer Modern are leading.

The AxisHeight, AccentBaseHeight and FlattenedAccentBaseHeight are set to the x-height, a value that is defined in all fonts. The SkewedFractionVerticalGap also gets that value. Other variables relate to the em-width (or \quad), for instance the SkewedFractionHorizontalGap that gets half that value. Of course these last two then assume that the engine handles skewed fractions.

Variables that directly map onto each other are StretchStackGapBelowMin as bigopspacing1, StretchStackTopShiftUp as bigopspacing3, StretchStackGapAboveMin as bigopspacing2 and StretchStackBottomShiftDown as bigopspacing4. However, these clash with UpperLimitBaselineRiseMin as bigopspacing3, UpperLimitGapMin as bigopspacing1, LowerLimitBaselineDropMin as bigopspacing4 and LowerLimitGapMin as bigopspacing2. Where in traditional fonts these are the same, in OpenType they can be different. Should they be?

Internally we use different names for variables, simply because the engine has some parameters that OpenType maths hasn't. So we have limit_above_kern and limit_below_kern for bigopspacing5.

A couple of parameters have different values for (cramped) displaystyle. The FractionDelimiterSize and FractionDelimiterDisplayStyleSize use delim2 and delim1. The FractionDenominatorShiftDown and FractionDenominatorDisplayStyleShiftDown map onto denom2 and denom1 and their numerator counterparts from num2 and num1. The Stack* parameters also use these. The sub1, sub2, sup1, sup2, sup3, and supdrop can populate the Sub* and Super* parameters, also in different styles.

The rest of the parameters can be defined in terms of the default rulethickness, quad or xheight, often multiplied by a factor. For some we see the 1/18 show up a number that we also see with muskips. Some constants can be set from registers, like SpaceAfterScript which is just \scriptspace.

If you look at the LuaTEX source you wil find a section where this mapping is done in the case of a traditional font, that is: one without a math constants table. In LuaMetaTEX we don't need to do this because font loading happens in Lua. So we simply issue an error when the math engine can't resolve a mandate parameter. The fact that we have a partial mapping from math constants onto traditional parameters and that LuaTEX has to deal with the traditional ones too make for a somewhat confusing landscape. When in LuaMetaTEX we assume wide fonts to be used that have a math constants table, we can probably clean up some of this.

We need to keep in mind that Cambria was the starting point, and it did borrow some concepts from TEX. But TEX had parameters because there was not enough information in the glyphs! Also, Cambria was meant for MS Word, and a word processor is unlikely to provide the level of control that TEX offers, so it needs some directions with respect to e.g. spacing. Without user control, it has to come up with acceptable compromises. So actually the LuaMetaTEX math engine can be made a bit cleaner when we just get rid of these parameters.

So, which constants are actually essential? The AxisHeight is important and also design related. Quite likely this is where the minus sits above the baseline. It is used for displacements of the baseline so that for instance fractions nicely align. When testing script anchored to fences we noticed that the parenthesis in XITS had too little depth while STIX had the expected amount. This relates to anchoring relative to the math axis.

Is there a reason why UnderbarRuleThickness and OverbarRuleThickness should differ? If not, then we only need a variable that somehow tells us what thickness fits best with the other top and bottom accents.

It is quite likely the same as the RadicalRuleThickness, which is needed to extend the radical symbol. So, here three constants can be replaced by one design related one. The FractionRuleThickness can also be derived from that, but more likely is that it is a quantity that the macro package sets up anyway, maybe related to rules used elsewhere.

The MinConnectorOverlap and RadicalDegreeBottomRaisePercent also are related to the design although one could abuse the top accent anchor for the second one. So they are important. However, given the small number of extensibles, they could have been part of the extensible recipes.

The AccentBaseHeight and FlattenedAccentBaseHeight might relate to the margin that the designer put below the accent as part of the glyph, so it is kind of a design related constant. Nevertheless, we fix quite some accents in the goodie files because they can be inconsistent. That makes these constants somewhat dubious too. If we have to check a font, we can just as well set up constants that we need in the goodie file. Also, isn't it weird that there are no bottom variants?

We can forget about MathLeading as it serves no purpose in TEX. The DisplayOperatorMinHeight is often set wrong so although we fix that in the goodie file it might be that we just can use an internal variable. It is not the font designer who decides that anyway. The same is true for DelimitedSubFormulaMinHeight.

If we handle skewed fractions, SkewedFractionHorizontalGap and SkewedFractionVerticalGap might give an indication of the tilt but why do we need two? It is design related though, so they have some importance, when set right.

The rest can be grouped, and basically we can replace them by a consistent set of engine parameters. We can still set them up per font, but at least we can then use a clean set. Currently, we already have more. For instance, why only SpaceAfterScript and not one for before, and how about prescripts and primes? If we have to complement them with additional ones and also fix them, we can as well set up all these script related variables.

For fractions the font provides FractionDenominatorDisplayStyleGapMin, FractionDenominatorDisplayStyleShiftDown, FractionDenominatorGapMin, FractionDenominatorShiftDown, FractionNumeratorDisplayStyleGapMin, FractionNumeratorDisplayStyleShiftUp, FractionNumeratorGapMin and FractionNumeratorShiftUp. We might try to come up with a simpler model.

Limits have: LowerLimitBaselineDropMin, LowerLimitGapMin, UpperLimitBaselineRiseMin and UpperLimitGapMin. Limits are tricky anyway as they also depend on abusing the italic correction for anchoring.

Horizontal bars are driven by OverbarExtraAscender, OverbarVerticalGap, UnderbarExtraDescender and UnderbarVerticalGap, but for e.g. arrows we are on our own, so again a not so useful set.

Then radicals: we need some more than these RadicalDisplayStyleVerticalGap, RadicalExtraAscender, RadicalKernAfterDegree, RadicalKernBeforeDegree and RadicalVerticalGap, and because we really need to check these there is no gain having them in the font.

Isn't it more a decision by the macro package how script and scriptscript should be scaled? Currently we listen to ScriptPercentScaleDown and ScriptScriptPercentScaleDown, but maybe it relates more to usage.

We need more control than just SpaceAfterScript and an engine could provide it more consistently. It's a loner.

How about StackBottomDisplayStyleShiftDown, StackBottomShiftDown, StackDisplayStyleGapMin, StackGapMin, StackTopDisplayStyleShiftUp and StackTopShiftUp? And isn't this more for the

renderer to decide: StretchStackBottomShiftDown, StretchStackGapAboveMin, StretchStackGapBelowMin and StretchStackTopShiftUp?

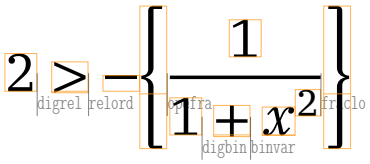
This messy bit can also be handled more convenient so what exactly is the relationship with the font design of SubSuperscriptGapMin, SubscriptBaselineDropMin, SubscriptShiftDown, SubscriptTopMax, SuperscriptBaselineDropMax, SuperscriptBottomMaxWithSubscript, SuperscriptBottomMin, SuperscriptShiftUp and SuperscriptShiftUpCramped?

Just for the record, here are the (font related) ones we added so far. A set of prime related constants similar to the script ones: PrimeRaisePercent, PrimeRaiseComposedPercent, PrimeShiftUp, PrimeBaselineDropMax, PrimeShiftUpCramped, PrimeSpaceAfter and PrimeWidthPercent. Of course, we also added SpaceBeforeScript just because we want to be symmetrical in the engine where we also have to deal with prescripts.

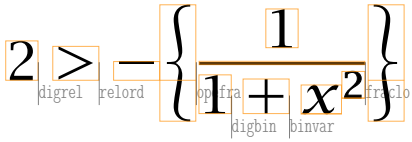
These we provide for some further limit positioning: NoLimitSupFactor and NoLimitSubFactor; these for delimiters: DelimiterPercent and DelimiterShortfall; and these for radicals in order to compensate for sloping shapes: RadicalKernAfterExtensible and RadicalKernBeforeExtensible because we have doublesided radicals.

Finally, there are quite some (horrible) accent tuning parameters: AccentTopShiftUp, AccentBottomShiftDown, FlattenedAccentTopShiftUp, FlattenedAccentBottomShiftDown, AccentBaseDepth, AccentFlattenedBaseDepth, AccentTopOvershoot, AccentBottomOvershoot, AccentSuperscriptDrop, AccentSuperscriptPercent and AccentExtendMargin, but we tend to move some of that to the tweaks on a per accent basis.

Setting these parameters right is not trivial, and also a bit subjective. We might, however, assume that for instance the math axis is set right, but alas, when we were fixing the less and greater symbols in Lucida Bright Math, we found that all symbols actually were designed for a math axis of 325, instead of the given value 313, and that difference can be seen!



Old Lucida



New Lucida

The assumption is that the axis goes trough the middle of the minus. Luckily it was relatively easy to fix these two symbols (they also had to be scaled, maybe they originate in the text font?) and adapt the axis. We still need to check all the other fonts, but it looks like they are okay, which is good because the math axis plays an important role in rendering math. It is one of the few parameters that has to be present and right. A nice side effect of this is that we end up with discussing new (ConTEXt) features. One can for instance shift all non-character symbols down just a little and lower the math axis, to get a bit more tolerance in lines with many inline fractions, radicals or superscripts, that otherwise would result in interline skips.

A first step in getting out of this mess is to define *all* these parameters in the goodie file where we fix them anyway. That way we are at least not dependent on changes in the font. We are not a word processor so we have way more freedom to control matters. And preset font parameters sometimes do more harm than good. A side effect of a cleanup can be that we get rid of the evolved mix of uppercase and lowercase math control variables and can be more consistent. Ever since LuaTEX got support for OpenType, math constants names have been mapped and matched to traditional TEX font parameters.

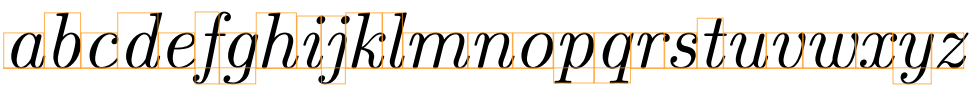
12.4 Metrics

With metrics we refer to the dimensions and other properties of math glyphs. The origin of digital math fonts is definitely Computer Modern and thereby the storage of properties is bound to the tfm file format. That format is binary and can be loaded fast. It can also be stored in the format, unless you're using LuaTEX or LuaMetaTEX where Lua is the storage format. A tfm file stores per character the width, height, depth and italic correction. The file also contains font parameters. In math fonts there are extensible recipes and there is information about next in size glyphs. The file has kerning and ligature tables too.

Given the times TEX evolved in, the format is rather compact. For instance, the height, depth and italic correction are shared and indices to three shared values are used. There can be 16 heights and depths and 64 italic corrections. That way much fits into a memory word.

The documentation tells us that "The italic correction of a character has two different uses. (a) In ordinary text, the italic correction is added to the width only if the TEX user specifies '\/' after the character. (b) In math formulas, the italic correction is always added to the width, except with respect to the positioning of subscripts." It is this last phenomena that gives us some trouble with fonts in OpenType math. The fact that traditional fonts cheat with the width and that we add and selectively remove or ignore the correction makes for fuzzy code in LuaTEX although splitting the code paths and providing options to control all this helps a bit. In LuaMetaTEX we have more control but also expect an OpenType font. In OpenType math there are italic corrections, and we even have the peculiar usage of it in positioning limits. However, the idea was that staircase kerns do the detailed relative positioning.

Before we dive into this a bit more, it is worth mentioning that Don Knuth paid a lot of attention to details. The italic alphabet in math uses the same shapes as the text italic but metrics are different as is shown below. We have also met fonts where it looked like the text italics were taken, and where the metrics were handled via more excessive italic correction, sometimes combined with staircase kerns that basically were corrections for the side bearing. This is why we always come back to Latin Modern and Cambria when we investigate fonts: one is based on the traditional TEX model, with carefully chosen italic corrections, and the other is based on the OpenType model with staircase kerning. They are our reference fonts.



Latin Modern Roman Italic

Latin Modern Math Italic

In ConTEXt MkIV we played a lot with italic correction in math and there were ways to enforce, ignore, selectively apply it, etc. But, because fonts actually demand a mixture, in LuaMetaTEX we ended up with more extensive runtime patching of them. Another reason for this was that math fonts can have weird properties. It looks like when these standards are set and fonts are made, the font makers can do as they like as long as the average formula comes out right, and metrics to some extent resemble a traditional font. However, when testing how well a font behaves in a real situation there can be all kind of interferences from the macro package: inter-atom kerning, spacing corrections macros, specific handling of cases, etc. We even see OpenType fonts that seem to have the same limited number of heights, depths and italic corrections. And, as a consequence we get for instance larger sizes of fences having the same depth for all the size variants, something that is pretty odd for an OpenType font with no limitations.

The italic correction in traditional TEX math gets added to the width. When a subscript is attached to a kernel character it sits tight against that character: its position is driven by the width of the kernel. A superscript on the other hand is moved over the italic width so that it doesn't overlap or touch the likely sticking out bit of the kernel. This means that a traditional font (and quite some OpenType math fonts are modelled after Computer Modern) have to find compromises of width and italic correction for characters where the subscript is supposed to move left (inside the bounding box of the kernel).

The OpenType specification has some vague remarks about applying italic correction between the last in a series of slanted shapes and operators, as well as positioning limits, and suggests that it relates to relative super- and subscript positioning. It doesn't mention that the correction is to be added to the width. However, the main mechanism for anchoring script are these top and bottom edge kerns. It's why in fonts that provide these, we are unlikely to find italic correction unless it is used for positioning limits.

It is for that reason that an engine can produce reasonable results for fonts that either provide italics or provide kerns for anchoring: having both on the same glyph would mean troubles. It means that we can configure the engine options to add italic correction as well as kerns, assuming distinctive usage of those features. For a font that uses both we need to make a choice (this is possible, since we can configure options per font). But that will never lead to always nicely typeset math. In fact, without tweaks many fonts will look right because in practice they use some mixture. But we are not aiming at partial success, we want all to look good.

Here is another thing to keep in mind (although now we are guessing a bit). There is a limited number of heights and depths in TEX fonts possible (16), but four times as many italic corrections can be defined (64). Is it because Don Knuth wanted to properly position the sub- and subscripts? Adding italic correction to the width is pretty safe: shapes should not overlap. Choosing the right width for a subscript needs more work because it's is more visual. In the end we have a width that is mostly driven by superscript placement! That also means that as soon as we remove the italic correction things start looking bad. In fact, because also upright math characters have italic correction the term 'italic' is a bit of a cheat: it's all about script positioning and has little to do with the slope of the shapes.

One of the reasons why for instance spacing between an italic shape and an upright one in TEX works out okay is that in most cases they come from a different font, which can be used as criterium for keeping the correction; between a sequence of same-font characters it gets removed. However, in OpenType math there is a good chance that all comes from the same font (at least in ConTEXt), unless one populates many families as in traditional TEX. We have no clue how other macro packages deal with this but it might well be the case that using many families (one for each alphabet) works better in the end. The engine is really shape and alphabet agnostic, but one can actually wonder if we should add a glyph property indicating the distinctive range. It would provide engine level control over a run of glyphs (like multiplying a variable represented by a greek alpha by another variable presented by an upright b).

But glyph properties cannot really be used here because we are still dealing with characters when the engine transforms the noad list into a node list. So, when we discussed this, we started wondering how the engine could know about a specific shape (and tilt) property at all, and that brought us to pondering about an additional axis of options. We already group characters in classes, but we can also group them with properties like `tilted`, `dotless`, `bold`. When we pair atoms we can apply options, spacing and such based on the specific class pair, and we can do something similar with category pairs. It basically boils down to for instance `\mccode` that binds a character to a category. Then we add a command like `\setmathcategorization` (analogue to `\setmathspacing`) that binds options to pairs of categories. An easier variant of this might be to let the `\mccode` carry a (bit)set of options that then get added to the already existing options that can be bound to character noads as we create them. This saves us some configuration. Deciding what suits best depends on what we want to do: the fact that TEX doesn't do this means that probably no one ever gave it

much thought, but once we do have this mechanism it might actually trigger demand, if only by staring at existing documents where characters of a different kind sit next to each other (take this 'a' invisible times 'x'). It would not be the first time that (in ConTEXt) the availability of some feature triggers creative (ab)usage.

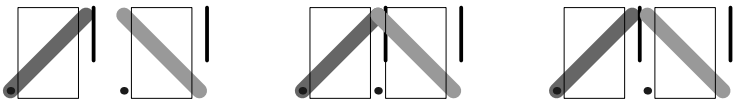
Because the landscape has settled, because we haven't seen much fundamental evolution in OpenType math, because in general TEX math doesn't really evolve, and because ConTEXt in the past has not been seen as suitable for math, we can, as mentioned before, basically decide what approach we follow. So, that is why we can pick up on this italic correction in a more drastic way: we can add the correction to the width, thereby creating a nicely bounded glyph, and moving the original correction to the right bottom kern, as that is something we already support. In fact, this feature is already available, we only had to add setting the right bottom kern. The good news is that we don't need to waste time on trying to get something extra in the font format, which is unlikely to happen anyway after two decades.

It is worth noticing that when we were exploring this as part of using MetaPost to analyze and visualize these aspects, we also reviewed the `wipeitalics` tweak and wondered if, in retrospect, it might be a dangerous one when applied to alphabets (for digits and blackboard bold letters it definitely makes sense): it can make traditional super- and subscript anchoring less optimal. However, for some fonts we found that improper bounding boxes can badly interfere anyway: for instance the upright 'f' in EBGaramond sticks out left and right, and has staircase kerns that make scripts overlap. The right top of the shape sticks out a lot and that is because the text font variant is used. We already decided to add a `moveitalics` tweak that moves italic kerns into the width and then setting a right bottom kern that compensates it that can be a pretty good starting point for our further exploration of optimal kerns at the corners. That tweak also fixes the side bearings (negative llx) and compensates left kerns (when present) accordingly. An additional `simplifykerns` tweak can later migrate staircase kerns to simple kerns.

So, does that free us from tweaks like `dimensions` and `kerns`? Not completely. But we can forget about the italic correction in most cases. We have to set up less lower right kerns and maybe correct a few. It is just a more natural solution. So how about these kerns that we need to define? After all, we also have to deal with proper top kerns, and like to add kerns that are not there simply because the mentioned comprise between width, italic and the combination was impossible. More about that in the next section.

12.5 Kerning

In the next pictures we will try to explain more visual what we have in mind and are experimenting with as we write this. In the traditional approach we have shapes that can communicate the width, height, depth and italic correction to the engine so that is what the engine can work with. The engine also has the challenge to anchor subscripts and superscripts in a visual pleasing way.



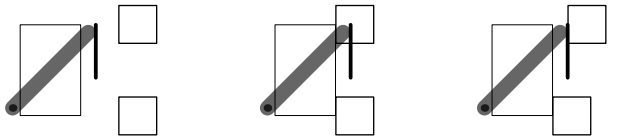
two characters width only with italic

In this graphic we show two pseudo characters. The shown bounding box indicates the width as seen by the engine. An example of such a shape is the math italic f, and as it is used a lot in formulas it is also one of the most hard ones to handle when it comes to spacing: in nearly all fonts the right top sticks out and in some fonts the left part also does that. Imagine how that works out with scripts, fences and preceding characters.

When we put two such characters together they will overlap, and this is why we need to add the italic correction. That is also why the TEX documentation speaks in terms of "always add the italic correction to the width". This also means that we need to remove it occasionally, something that you will notice when you

study for instance the LuaTEX source, that has a mix of traditional and OpenType code paths. Actually, compensating can either be done by changing the width property of a glyph node or by explicitly adding a kern. In LuaMetaTEX we always add real kerns because we can then trace better.

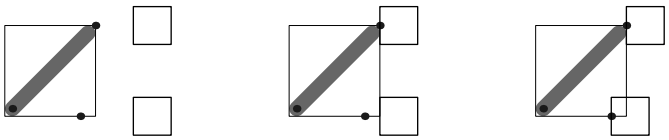
The last graphic in the above set shows how we compensate the width for the bit that sticks out. It also shows that we definitely need to take neighboring shapes into account when we determine the width and italic correction, especially when the later is *not* applied (read: removed).



Here we anchored a super- and subscript. The subscript position it tight to the advance width, again indicated by the box. The superscript however is moved by the italic correction and in the engine additional spacing before and after can be applied as well, but we leave that for now. It will be clear that when the font designer chooses the width and italic correction, the fact that scripts get attached has to be taken into account.

In this graphic we combine the italic correction with the width. Keep in mind that in these examples we use tight values but in practice that correction can also add some extra right side bearing (white space). This addition is an operation that we can do when loading a font. At the same time we also compensate the left edge for which we can use the x coordinate of the left corner of the glyphs real bounding box. The advance width starts at zero and that corner is then left of the origin. By looking at shapes we concluded that in most cases that shift is valid for usage in math where we don't need that visual overlap. In fact, when we tested some of that we found that the results can be quite horrible when you don't do that; not all fonts have left bottom kerning implemented.

The dot at the right is actually indicating the old italic correction. Here we let it sit on the edge but as mentioned there can be additional (or maybe less) italic correction than tight.

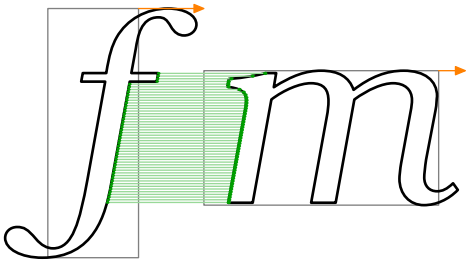


Finally we add the scripts here. This time we position the superscript and subscript at the top and bottom anchors. The bottom anchor is, as mentioned, the old italic correction, and the top one currently just the edge. And this is what our next project is about: identify the ideal anchors and use these instead.

In the ConTEXt goodie files (the files that tweak the math fonts runtime) we can actually already set these top and bottom anchors and the engine will use them when set. These kerns are not to be confused with the more complicated staircase kerns. They are much simpler and lightweight. The fact that we already have them makes it relatively easy to experiment with this.

It must be noted that we talk about three kinds of kerns: inter character kerns, corner kerns and staircase kerns. We can set them all up with tweaks but so far we only did that for the most significant ones, like integrals. The question is: can we automate this? We should be careful because the bad top accent anchors in the TEX Gyre fonts demonstrate how flawed heuristics can be. Interesting is that the developers of these font used MetaPost and are highly qualified in that area. And for us using MetaPost is also natural!

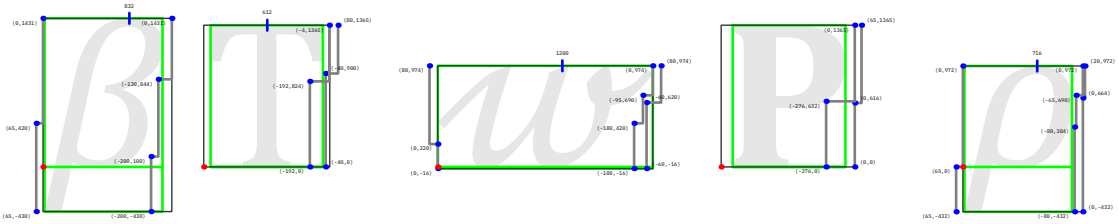
The approach that we follow is somewhat interactive. When working on the math update we like to chat (with zoom) about these matters. We discuss and explore plenty and with these kerns we do the same. Because MetaPost produces such nice and crispy graphics, and because MetaFun is well integrated into ConTEXt we can link all these subsystems and just look at what we get. A lot is about visualization: if we discuss so called 'grayness' in the perspective of kerning, we end up with calculating areas, then look at what it tells us and as a next step figure out some heuristic. And of course we challenge each other into new trickery.



We are sure that getting this next stage in the perfection of math typesetting in ConTEXt and LuaMetaTEX will take quite some time, but the good news is that all machinery is in place. We also have to admit that it all might not work out well, so that we stick to what we have now. But at least we had the fun then. And it is also a nice example of both applying mathematics and programming graphics.

That said, if it works out well, we can populate the goodie files with output from MetaPost, tweak a little when needed, and that saves us some time. One danger is that when we try to improve rendering the whole system also evolves which in turn will give different output, but we can always implement all this as features because after all ConTEXt is very much about configuration. And it makes nice topics for articles and talks too!

The kerns discussed in the previous paragraphs are not the ones that we find in OpenType fonts. There we have 'staircase' kerns that stepwise go up or down by height and kern. So, one can have different kerns depending on the height and sort of follow the shape. This permits quite precise kerning between for instance the right bottom of a kernel and left top of a subscript. So how is that used in practice? The reference font Cambria has these kerns but close inspection shows that these are not that accurate. Fortunately, we never enter the danger zone with subscripts, because other parameters prevent that. If we look at for instance Lucida and Garamond, then we see that their kerns are mostly used as side bearing, and not really as staircase kerns.



U+1D6FD U+003A4 U+1D4CC U+1D6B8 U+1D70C

In these figures you see a few glyphs from cambria with staircase kerns and although we show them small you will notice that some kern boundaries touch the shape. As subscripts never go that high it goes unnoticed but it also shows that sticking to the lowest boundary makes sense.

We conclude that we can simplify these kerns, and just transform them into our (upto four) corner kerns. It is unlikely that Cambria gets updates and that other fonts become more advanced. One can even wonder if multiple steps really give better results. The risk of overlap increases with more granularity because not every pair of glyphs is checked. Also, the repertoire of math characters will likely not grow and include shapes that differ much from what we can look at now. Reducing these kerns to simple ones, that can easily be patched at will in a goodie file, has advantages. We can even simplify the engine.

12.6 Conclusion

So how can we summarize the above? The first conclusion is that we can only get good results when we runtime patch fonts to suite the engine and our (ConTEXt) need. The second conclusion is that we should seriously consider to drop (read: ignore) most math font parameter and/or to reorganize them. There is no need to be conforming, because these parameters are often not that well implemented (thumb in mouth). The third conclusion (or observation) is that we should get rid of the excessive use of italic correction, and go for our new corner kerns instead. Last, we can conclude that it makes sense to explore how we can use MetaPost to analyze the shapes in such a way that we can improve inter character kerning, corner kerns and maybe even, in a limited way, staircase kerns.

And, to come back to accents: very few characters need a top kern. Most can be handled with centered anchors, and we need tweaks for margins and overshoot anyway. The same is true for many other tweaks: they are there to stay.

This is how we plan to go forward:

- We pass no italic corrections in the math fonts to the engine, but instead we have four dedicated simple corner kerns, top and bottom anchors, and we also compensate negative left side bearing. We should have gone that route earlier (as follow up on a MkIV feature) but were still in some backward compatibility mindset.
- The LuaMetaTEX math engine might then be simplified by removing all code related to italic correction. Of course it hurts that we spent so much time on that over the years. We can anyway disable engine options related to italic correction in the ConTEXt setup. Of course the engine is less old school generic then but that is the price of progress.
- A default goodie file is applied that takes care of this when no goodie file is provided. We could do some in the engine, but there is no real need for that. We can simplify the mid 2022 goodie files because we have to fix less glyphs.
- If we ever need italic correction (that is: backtrack) then we use the (new) `\mccode` option code that can identity sloped shapes. But, given that ignoring the correction between sloped shapes looks pretty bad, we can as well forget about this. After all, italic correction never really was about correcting italics, but more about anchoring scripts.
- Staircase kerns can be reduced to simple corner kerns and the engine can be simplified a bit more. In the end, all we need is true widths and simple corner kerns.
- We reorganize the math parameters and get rid of those that are not really font design dependent. This also removes a bit of overlap. This will be done as we document.
- Eventually we can remove tweaks that are no longer needed in the new setup, which is a good thing as it also save us some documenting and maintenance.

All this will happen in the perspective of ConTEXt and LuaMetaTEX but we expect that after a few years of usage we can with confidence come to some conclusions that can trickle back in the other engines so that other macro packages can benefit from a somewhat radical different but reliable approach to math rendering, one that works well with the old and new fonts.

13 Gaining performance

In the meantime (2022) the LuaMetaTEX engine has touched many aspects of the original TEX implementation. This has resulted in less memory consumption than for instance LuaTEX when we talk tokens, more efficient macro handing, additional storage options and numerous new features and optimizations. Of course one can disagree about all of this, but what matters to us is that it facilitates ConTEXt well. That macro package went from MkII to MkIV to MkXL (aka LMTX).

Although over the years the macros evolved the basic ideas haven't changed: it is a keyword driven macro package that is set up in a way that makes it possible to move forward. In spite of what one might think, the fundamentals didn't change much. It looks like we made the right decisions at the start, which means that we can change low level implementations to match the engine without users noticing much. Of course in the area of fonts, input encoding and languages things have changed simply because the environment in which we operate changes.

A fundamental difference between pdfTEX and LuaMetaTEX is that the later is in many aspects 32 and even 64 bit all over the place. That comes with a huge performance hit but also with possibilities (that I won't discuss here now)! On a simple document nothing can beat pdfTEX, even with the optimizations that we can apply when using the modern engines. However, on more complex documents reality is that LuaMetaTEX can outperform pdfTEX, and documents (read: user demands) have become more complex indeed.

So, how does that work in practice? One can add some features to an engine but then the macro package has to be adapted. Due to the way ConTEXt is organized it was not that hard to keep it in sync with new features, although not all are applied yet to full extend. Some new features improved performance, others made the machinery (or its usage) a bit slower. The first versions of LuaMetaTEX were some 25% slower than LuaTEX, simply because the backend is written in Lua. But, end 2022 we can safely say that LuaMetaTEX can be 50% faster than its ancestor. This is due to a mix of the already mentioned optimizations and new features, for instance a more powerful macro parser. The backend has become more complex too, but also benefits from a few more helpers.

Because we spend a lot of time in Lua the interfaces to TEX have been extended and improved too. Of course we depend on the Lua interpreter being kept in optimum state by its authors. It must be said that quite some of the interfaces might look obscure but these are not really meant for the average user anyway. Also, as soon as one messes with tokens and nodes at that level one definitely need to know what one's doing!

The more stable the engine becomes, the less there is to improve. Occasionally it was possible to squeeze our a few more milliseconds on run but it depends a lot of what one does. And TEX is already quite fast anyway. Of course 0.005 seconds on a 5 second run is not much but hundred times such an improvement is noticeable, especially when there are multiple runs or when one processes a batch of 10.000 documents (each needing two runs).

One interesting aspect of TEX that it can surprise you every now and then. End 2022 I decided to play a bit more with a feature that has been around for a while:

```
\integerdef  \fooA 123
\dimensiondef\fooB 123pt
```

These primitives create a counter and a dimen where the value is stored in the hash table. The original reason was that I didn't want to spoil registers. But although these are basically constants there is more to it now.

```
\countdef\fooC 27
\dimendef\fooD 56
```

These primitives create a command that stores the register number (here 27 and 56) with the name. In this case a 'variable' is accessed in two steps: the \fooC macro expands to an register accessor with value 27. Next that accessor will kick in and fetch (or set) the value in slot 27 of the memory range bound to (in total 65K) counters. All these registers sit a the lower end of TEX's memory which is definitely not next to the meaning of \fooC. So we have two memory accesses to get to the number. Contrary to that once we are at \fooA we are also at the value. Although memory access can be fast when the relevant slots are cached in practice it can give delays, especially in a program like TEX where most data is spread all over the place. And imagine other processes competing for access too.

It is for that reason that I decided to replace the more or less 'constant' property of \fooA by one that also supports assignments As well as the arithmic commands like \advance. This was not that hard due to the way the LuaMetaTEX source is organized. After that using these pseudo constants proved to be more efficient than registers, but of course I then had to adapt the source. Interestingly that should have been easy because one only needs to change the definitions of for instance \newcount but in practice that doesn't work because it will/can break for instance generic packages like Tikz.

So, in the end a new allocator was added and just over 1000 lines in some 120 files (with some overlap) had to be adapted to this. In addition some precautions had to be made for access from Lua because the quantities were no longer registers. But it was rewarding in the sense that the test suite now ran some 5% faster and processing the LuaMetaTEX manual went from 8.7 seconds on my laptop down to around 8.5, which is not bad.

Now why do we bother so much about performance? If I really want a faster run using a decent desktop is of more help. But even then there can be reasons. When Mikael and I were discussing math engine developments at some point we noticed that a run took twice as much time as a result of (supposedly idle) background tasks. Now keep in mind that TEX uses a single core so with plenty cores it should not be that bad. However, when the video chat program takes half of the CPU power, or when a mathematical manipulation program idles in the background taking 80 percent of a modern machine, or when a popular editor keeps all kind of plug ins busy for no reason, or when a supposedly closed a browser consumes gigabytes of memory and keeps dozens of supposedly idle threads busy, it becomes clear that we should not let TEX put a large burden on memory access (and cache).

It can get even worse when one runs on virtual machines where the host suggests that you get 16 cores so that you can run a dozen TEX jobs in parallel but simple measurements show that these shared cores report a much higher ideal performance than the one you measure. So, the less demanding a ConTEXt run becomes, the better: we're not so much after the .2 seconds on a 8 second run, but more after 3 seconds for that same run when using shared resources where it became 15 seconds. And this is what observations with respect to the performance of the test suite seem to indicate.

In the end it's mostly about comfort: when you process a document of 300 pages, 10 seconds is quite okay for a few changes, because one can relate time to output, but 20 seconds ... And when processing a a few page document the waiting time of a second is often less than what one needs to move the mouse around to the viewer. Also, when a user starts TEX on the console and afterwards opens a browser from there that second is even less noticeable.

Now let's go back to improvements. A related addition was \advanceby that doesn't check for the by keyword. When there is no such keyword we can avoid pushing back the non-matching next token which is also noticeable. Here about 680 changes were needed. Changes like these only make a difference in performance for some very demanding mechanisms in ConTEXt. Again one cannot overload an existing primitive

because generic packages can fail (as the test suite proved). There were also a few places where a dirty trick had to be changed because we cannot alias these constants.

We can give similar stories about other improvements but this one sort of stands out because it is so noticeable. Also, other changes involve more drastic low level adaptations of ConTEXt so these happen over a longer period of time. Of course all has to happen in ways that don't impact users. An example of a performance primitive is `\advancebyplusone` which is actually implemented but still disabled because the gain is in hundreds of seconds range and I need to (again) adapt the source in order to benefit.

The mentioned register variants are implemented for count (integer), dimen (dimension), skip (gluespec) and muskip (mugluespec). Token registers are more complex as they have reference counters as well as more manipulator primitives. The same is true for boxes (although it is tempting to come up with some faster access mechanism) and attributes, that also have more diverse accessors. Also, token lists and boxes involve way more than a simple assignment or access so any gain will drown in other actions. That said, it really makes sense now to drop the maximum of 64K registers to some more reasonable 8K (or even less for mu skips). That will save a couple of megabytes which sounds like little but still puts less burden on the system.

14 LMTX on a phone

When my FairPhone 2 started to get issues (running hot and then rebooting) and some spare parts became hard to get, I moved on to a FairPhone 4. We're talking early 2022. The specifications of that little computer, which comes with a 5 year warrantee and long term support are quite okay: a 1080x2340 pixel display, a Qualcomm SM7225 Snapdragon 750G (Octa-core (2x2.2 GHz Kryo 570 & 6x1.8 GHz Kryo 570), an Adreno 619 GPU, 8GB memory. an 256GB solid state disk, the usual phone gadgets like audio, camera, wireless, bluetooth and gps, and an USB Type-C 3.0 connector with support for OTG and DisplayPort.

Why do these specification matter? One reason is that in the compile farm we generate binaries for ARM processors and this phone has a decent one. The fast cores are in the same league as an over-clocked RaspberryPi 4 that we use in the compile farm for generating 32 bit binaries; the 64 bit binaries are generated in a virtual machine on a Mac Mini. So, in 2023, when looking at that phone, I wondered if we could run LMTX on it. I installed the UserLand linux stub from the Android Playstore and got myself an Ubuntu headless installation. After downloading the LMTX installer indeed I could install the distribution on the little machine.

A next step was trying to connect the phone to the display on my desk and after getting the right USB-C cable from the local computer shop I managed to get a bit larger terminal although Android 12 seems not able to use the whole 4K screen. Putting it in developers mode made it possible to enable the Android desktop interface in an external monitor. A bluetooth keyboard and mouse completed the setup. Later I tried a linux desktop but that was quite a disappointment so more research is needed there.

A predictable next step was to see if I could compile the LuaMetaTEX source that is part of the installation. Installing gcc and cmake was easy and indeed compilation went pretty well after that.

A quick performance test showed that making a format, which includes generating the file database, initially takes 10 seconds but less that 4 seconds once files are cached. Processing 1000 paragraphs from the `tufte` sample file is done with a reasonable 55 pages per second. I didn't test more complex documents but that might happen later, when the dock that I ordered has arrived, and when I have a decent display setup.

Given the fact that I only use a handful of applications on the laptop one can wonder when the moment is there that a properly dockable phone can do the job. Of course a disadvantage is that batteries are too small so one needs to provide power, but one needs a monitor, keyboard and mouse anyway. Wear and tear of the ssd can also be an issue but when storage is plenty that should work out all right. Of course it also assumes a stable operating system with one's favourite editing platform and viewer available.

15 Running green

There are a few contradicting developments going on: energy prices sky-rocket and Intel and AMD are competing for the fastest cpu's where saving energy seems mostly related to making sure that the many cores running at the same time don't burn the machine. However, TEX is a single core consumer so throwing lots of cores into the game is not helping much. You're better served with one very fast core than many slower ones that accumulate to much horsepower. The later makes sense when you process video or play games, but that's not what TEX is about, although it is fun to play with. Of course often multiple cores come in handy, for instance in the build farm that is used to compile LuaMetaTEX and intermediate TEXLive releases: when that gets compiled and we also trigger a LuaMetaTEX build, two times 10 linux virtual machines are compiling and one windows machine that runs four compile jobs at the same time.

The server that runs the farm is Dell 710 server with dual 5630 Xeon processors, 6 SAS drives each 2GB in (hardware) raid 10, 72 GB memory, redundant power supplies and 6 network ports. It sits idle for most of the time and consumes between 250 and 400W. It is part of a redundant setup: dual switches, dual routers, multiple UPS's, air conditioning, two backup QNAP NAS's, a few low power machines for distributed continuous incremental backups, etc. The server itself is a refurbished one, so not the most expensive, but with the Dutch energy prices of 2022 bound to gas prices, we quickly realized that there was no way we could keep it up and running. Because we have three such servers (one is turned off and used as fallback) we started wondering if we could go for a different solution.

As we recently upgraded the 2013 laptops to refurbished 2018 ones (the latest models that could use the docking stations that we have), we decided to buy a few more and test these as replacements for the servers. Of course one has to pimp these machines a bit: a professional 2TB nvme SSD plus a proper 2.5in SSD as backup one, 64 GB of memory, a few extra USB3 network cards. The cpu's are fast mobile Xeons. We use proxmox as virtual host and that runs fine in such a configuration.

Surprisingly, after moving the farm to that setup, which basically boils down to moving virtual machines, we found that running those parallel compilations performance wise was quite okay. And the nice thing was that these machines idle much lower, some 20–30W. The saving is therefore quite noticeable and we decided to check some more; after all it would be nice if we could bring down the average power consumption of 1750W down to at least half so that it would match the output of a few solar panels. Of course it means that one has to ditch perfectly well working machines which itself is not that environmental friendly but there is not much to choose here.

The second machine to be replaced was the one that runs quite some virtual machines too: the main file server, the mail server, an ftp server, the website, an rsync host, the squeezebox server that also serves as update test, and various project related rendering services. All run in their own (OpenSuse) virtual machine. After installing a similar laptop those were also moved.

As a side effect, the two backup NAS's were replaced by a single laptop (my 2013 Dell precision workhorse) running one backup file server, and for an extra incremental backup (rsnaphot running hourly, daily, weekly and monthly backups is our friend) a 2013 macbook was turned into a linux machine (15W idle with an internal reused SSD[21] and an external 4GB disk), two managed switches became one (after all we had less network cables due to lost redundancy), only one backup power supply (that will be replaced by an nicer alternative when it breaks down; after all, by using laptops we get power backup for free). The total consumption went down with at least 1000W. Of course there is an investment involved and we need to reconfigure the server rack, but the expectation is that by investing now we get less troubles later (less gambling on energy).[22]

[21] For a change that apple machine was easy to update, and we could even get a new clone battery replacement.

But, there is still the pending question of what the impact is on the services that we run. The most demanding ones are the Math4all and Math4mbo: these produce large files, need many resources (xml and images), and we didn't want to burn ourselves too much. Now, here is an interesting observation: this service runs twice as fast on the new infrastructure. But it is hard to explain why. The file server is on a different machine (so no fast internal network), the cpu is a bit faster but not that much, the virtual machine is on ssd, but files are saved on the file server, which is a two disk usb3 enclosure connected directly to a virtual machine that does software raid. The most important difference is that main memory is much faster and TEX is a memory intense process. From when we started with LuaTEX we do know that memory bandwidth and cpu caches makes a difference. Maybe the faster floating point handling fo the more modern Xeon also helps here.

And that brings me to the following: how do we actually benchmark TEX? When you go on the internet and compare cpu's most tests are not that comparable to a TEX run on a single core. One can think of a set of test files, but the problem there is that when the engine evolves and details in the macro package coding changes, one looses the comparison with older tests. This is why, when we do such tests, we always run the same test on the different platforms. Although this often shows that the gain on newer hardware is seldom what one expects from the more general benchmarks, one can still be surprised. When we moved to five year newer laptops the gain was some 30% for me and 50% for my colleague. The difference between his laptop and the slightly more beefed up virtual machine can be neglected.

We monitor the power consumption with a youless device connected to the power meter. When I process the LuaMetaTEX manual I see the phase that the machine sits on go up 20W for a run that takes some 9 seconds. Let's say that we use 180Ws or 0.0006kWh (20.000 runs per kWh). So, compared to the idle power usage of a server, a single TEX run can be neglected, simply because it is so fast. So, what is actually the most efficient hardware for a TEX service? I get the feeling that a decent Intel Atom C3955 16-Core driven machine is quite okay for that, but I don't have that at hand and last time I checked one could not order anything anyway. And with prices of hardware going up it's also not something you try for fun. As comparison to what we have now, testing TEX on an Intel NUC11ATKC2 could also be interesting (it has an N4505 cpu). There was a time when I considered a bunch of raspberry pi's but they no longer are that cheap, given that you can get them, and adding a case and proper disc enclosure also adds up. When wrapped in a nice package the pi will probably a couple of times slower but it then probably also uses less power. These fitlets are also interesting but again, one can't get them.

It is kind of fun to play with optimizations that don't really impact the clarity of the code. One can argue that spending a day on something that saves 0.005 seconds on a specific run is a waste of time, but of course one has to multiply that number by a number of runs. Personally I will never gain from it but nevertheless it can save some energy: imagine a batch of 15000 documents every day. We then save $15000_{\text{highin}} * _{\text{highin}} 0.005_{\text{highin}} * 365_{\text{figure}} = _{\text{figure}} 27375$ seconds or about 8 hours runtime. This can still be neglected but what if this is not the only optimization?

An example of such an optimization is this:

```
\advance\somecounter    \plusone
\advance\somecounter by \plusone
```

The second one runs faster because there is no push back involved as side effect of the lack of a keyword, so how about adding this to the engine?

[22] We hope to save some 9000 kWh which means that save at least some 2500€ per year and more when the government will reinstate its energy tax policy and or prices go further up, which seems to be the case. Even before the crisis in the Netherlands 5ct/Kwh became fives times that amount effectively when connection, transportation, energy tax and value added tax gets added.

```
\advanceby   \somecounter \plusone
\advancebyone\somecounter
```

Given the way LuaMetaTEX is coded, it only needs a few lines! In this case it extends the repertoire of primitives so it is visible but we have many other (similarly small) optimizations that contribute. Again, the average user will not notice a drop in runtime from 1.5 seconds to 1.45 but when 8 hours become 80 hours or 800 hours it does become interesting. In energy sensitive 2022 these 800 hours not only save some €400 but also contribute to a lower carbon footprint! And now imagine how much could be saved on these extensive runs when we make sure that the style used is optimal? Of course, when we need two runs per document it starts adding up more.

Some experiments with a demanding file showed one percent gain (on a 2.7 seconds run) using the alternative integers, dimensions and advance primitives. However, using ConTEXt's compact font mode brought down runtime to 2.0 seconds! So, in the end it's all very relative. It is worth noticing that the .7 seconds saved on fonts is sort of constant, which means that accumulated gains elsewhere makes that .7 seconds more significant as we progress.

16 Supporting math in the JMN collection

Hans Hagen, Hasselt NL
Mikael Sundqvist, Lund SV

Introduction

In 2022 we overhauled math font support in ConTEXt, using new functionality in the LuaMetaTEX engine. By that time it had become clear that the OpenType math font landscape had more or less settled. The Latin Modern fonts as well as the TEXgyre fonts don't evolve, so we consider them being frozen. It is also unlikely that the reference Cambria font will change or become more complete. More recent math font are modeled as a mixture of Cambria and Latin Modern.

When we started with LuaTEX in ConTEXt we immediately started using Unicode math but the lack of proper Unicode fonts, with the exception of Cambria, resulted in creating virtual Unicode math fonts on the fly using the virtual font features of the LuaTEX. But when the OpenType math fonts came available that kind of trickery was no longer needed (or at least less preferred).

That is why we considered dropping the virtual math font mechanism from LMTX. We had already dropped `tx` (we can use Termes) and `px` (better use Pagella) fonts as well as Type1 based Latin Modern. Dropping the commercial Math Times was a logical next step, also because it has never been tested. The mixtures of Pagella and Euler were already replaced by using the upgraded tweak mechanism.

That left us with Antykwa, Iwona and Kurier, the fonts that the late Janusz Nowacky vectorized and that came with plenty OpenType text fonts but also with Type1 math companions. And, as we like these fonts, it meant that we had to come up with a solution. One option was to create proper OpenType math fonts, but another was to strip down the virtual math font mechanism to just support these fonts. There is some charm in keeping the Type1 fonts, also because it is a test case for (by now sort of obsolete) tfm metric, pfb outline, encoding and map files, for which we have code embedded so having a proper test case makes sense.

In the end we opted for the second solution so this is what the next sections are about: supporting OpenType math using Type1 fonts. We admit that it took way more time than a conversion to OpenType math fonts would have, but that is partly due to the fact that these fonts, and especially Antykwa, have some characteristic features that we wanted to use. So, in a sense it was also an esthetics challenge. It also helped that the font was used in realistic and moderately complex math rendering. We also note that rendering in LMTX is different (and hopefully better) than in MkIV because we try to benefit from the upgraded math engine in LuaMetaTEX.

This exploration is dedicated to Janusz who was one of the characteristic presenters of fonts at BachoTEX meetings, who contributed these fonts, and who in some sense was thereby kick starting the Polish TEX related font projects.

Virtual math

We keep this expose simple and only tell what we did, you can look at the `lfg` files and source code to see what magick is done. For the standard Antykwa we load `LatinModern-Math` first, so we cover all symbols that matter. Next we stepwise load `rm-anttr.tfm`, `mi-anttri.tfm`, `mi-anttbi.tfm`, `rm-anttb.tfm`, `sy-anttrz.tfm`, `ex-anttr.tfm`. For each we specify an encoding vector. Some are loaded multiple times with different vectors. Because we don't like the slanted curly braces we even load `AntykwaTorunska-Regular` in order to get the upright ones. This method is not that different from what we do in MkIV.

The specification of a loaded font also can contain a list of (named) characters that should be ignored, That was one of the new features in the virtual constructor. We take the math parameters from the fonts where these are specified, here in the symbols and extension fonts.

The fonts contain extra snippets of extensibles that one can use to construct some of these vertical and horizontal stretched symbols on the fly in addition to what the font metrics already define. Unfortunately some snippets are missing, like the six pieces that could make up horizontal and vertical bars, for which we now need to cheat. We considered making a companion font but for now we are in 'as good as we can' emulation mode.

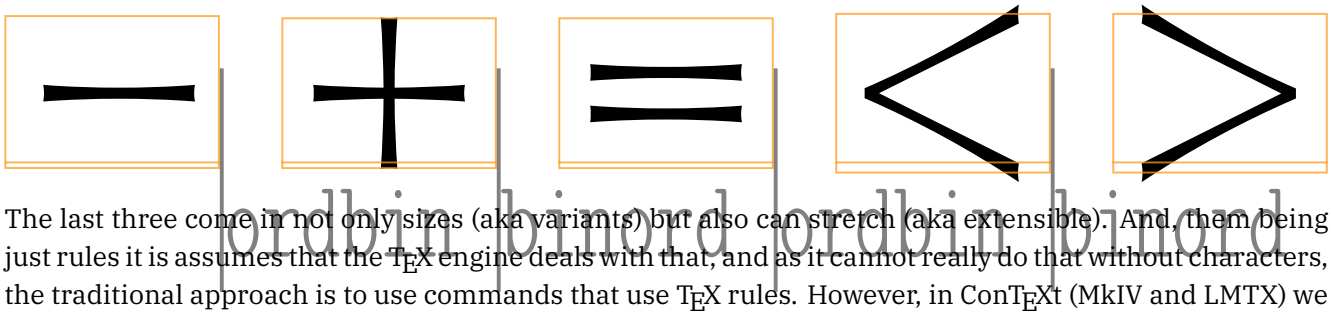
The fonts

We start out with a skeleton font and in the past we used the OpenType text font for that. On top of that we overlay a bunch of Type1 fonts, and as was common in those days, the ams math symbol fonts `msa` and `msb` were overlayed last in order to fill in remaining gaps. However, now that we have a Latin Modern OpenType math font it made more sense to use that as starting point because it already has all these symbols.

If we forget about the additional weights and condensed variants, the JMN math collection has actually not that many fonts. One reason for that is that the upright roman font, the ones that have an `r` near the end of the file name, in traditional TEX speak `rm`, have way more than 255 characters: it not only has all kind of composed characters, it also has all the extensible shapes. All is (as usual with Type1 fonts) driven by encoding and mapping files. Fortunately the glyphs names (that we use for filtering) are the same for the three fonts but there are some more in Antykwa.

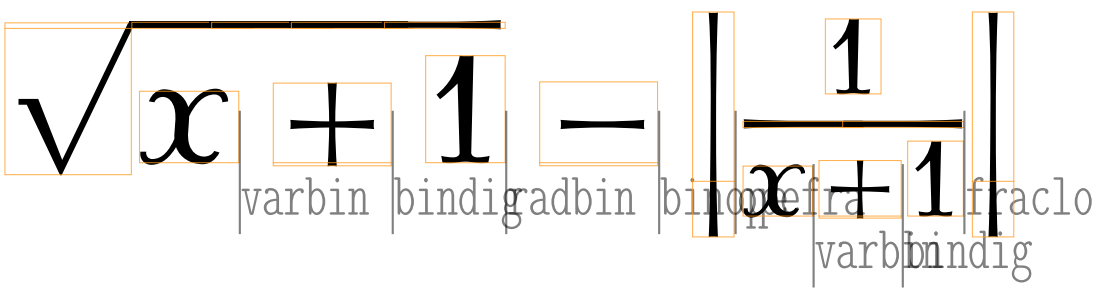
Challenges

The real challenge was Antykwa. This is because it has a distinctive curvature at the end of sticky parts (like rules and such). The TEX machinery as well as OpenType math assume rules being used in for instance radicals, fractions, overbars, underbars and vertical bar fences.



The last three come in not only sizes (aka variants) but also can stretch (aka extensible). And, them being just rules it is assumes that the TEX engine deals with that, and as it cannot really do that without characters, the traditional approach is to use commands that use TEX rules. However, in ConTEXt (MkIV and LMTX) we can provide the proper variants and extensible using virtual shapes, and in LMTX we can even scale as last resort.

The first two are special. We already could support fractions using dedicated characters because we played with the fraction builder using for instance arrows as separator and these are not rules but characters. It was not that much work to also make that possible for the rule in a radical. The main adaptation was that we need to center the numerator and denominator of a fraction and the body of a radical when the character used is wider than requested.



Because we have two font models in ConTEXt, normal and compact, we had to be careful in defining the virtual shapes and extensibles so that they work in both models. This has to do with scaling and sharing.

Implementation

The original virtual math font mechanism worked closely with the math fall back features, but these have been replaced by tweaks, which means that we lost some of that. Also, heuristics worked fine on the average but for Antykwa we wanted more. Therefore some of the built in logic has been moved to the goodie file that controls the composition. After all, we don't want to hard code specific solutions in the core.

Another addition was the use of so called setups bound to a font class so that we can set up some math machinery features for e.g. Antykwa in the goodie file. We need to bind to a class because we mix a dozen math fonts in one test file and therefore we need to separate these setups.

As in regular OpenType math support we ignore the italic correction and translate it in combinations of proper width and specific kerning. That way we avoid all kind of issues that we otherwise need to compensate for.

Because we need to hook extensible characters into the machine for Antykwa its font typescript file also defines a font class specific setup (of a few lines) to be applied. This might evolve into a more granular mechanism but for now it works fine and adds little overhead.

Examples

We end by showing a few "real" examples.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n.$$

The parentheses are unchanged, and we believe that using rotation symmetry instead of mirror symmetry is a brave but interesting choice, and the new fraction bar fits well with it. The fraction bar also fits well with the equal sign.

$$1 + x + x^2 + \ldots + x^n = \frac{1 - x^{n+1}}{1 - x}$$

The new vertical bars go well with the brackets, the integral and the solidus.

$$||f||_p = \left[\int_0^1 |f(x)|^p dx \right]^{1/p}$$

The fancy fraction bar and the radical bar have made the arithmetic-geometric mean inequality look more appealing than ever, hasn't it?

$$\frac{a_1 + a_2 + \ldots + a_n}{n} \geq \sqrt[n]{a_1 a_2 \ldots a_n}$$

Primes are usually a bit of a challenge:

$$\begin{aligned} f(k+1) - f(k) &= f'(k) + f''(k)\frac{1}{2} + f'''(\xi_k)\frac{1}{6} \\ &= k^p + \frac{p}{2}k^{p-1} + \frac{p(p-1)}{6}\xi_k^{p-2} \end{aligned}$$

And, as expected, multi-line formulas also look fine.

17 Profiles

17.1 Introduction

Among the typesetting problems that relate to math are inline formulas that have a bit too much height or depth but not so much as to justify some additional interline space. For that reason in ConTEXt MkII we have some snapping features that can be enabled that limit the dimensions. In MkIV a more extensive profile feature was written (we talked about it at meetings in 2015) that look at the bottom and top of lines in order to determine if lines can be moved closer, but in practice snapping and profiling are never really used. In the end it was more an academic exercise which is not uncommon when it comes to TEX user demands and practice.

As part of exploring math micro typography these features surfaced again during some discussion about weird mechanisms and we actually wondered if we could revive this now that we also control other aspects of math typesetting in more detail. One condition is that the overhead is not that high. Users accept some overhead for protrusion and expansion that relate to horizontal optimization so a little extra overhead for vertical optimization should not be a problem.

Of course, as with protrusion and especially expansion, the question is if readers will notice it. Best would be to set up some experiments but, although one can argue that research is important, in practice it always boils down to a visual impression, feel good and, like it or not, exploration, trial and error. And so a simplified variant of profiling was implemented and applied to a math intensive math book used in academia. Instead of proper experiments some unaware bystanders were asked if they noticed a difference and to our surprise that was the case! And these were not even math students but kids who were more familiar with children books and phones. That convinced us that we were on the right track, that we need to explain a little about what we actually do, and that we should tell users what to look at when this gets applied.

With an introduction like this, mentioning ‘research’, ‘academia’, ‘students’ and ‘typography’ we’re sure that future generations will be convinced that what is discussed next has a strong fundament so here we go!

17.2 A first example

We start with a simple example. Because in practice profiling always kicks in when possible one really need to handcraft an example that can be used for demonstration: figure 17.1.

In order to see what happens it is important to understand how TEX sees lines. Actually, the concept of lines in TEX is rather limited: lines are just horizontal boxes where the baselines are separated by `\baselineskip` and when the distance is larger than that dimensions a `\lineskip` gets added.

these are a few
lines of text
where lines have depths
or no real
height at all

Between all these lines some skip needs to be added: the `\baselineskip` minus the height and depth. If we add struts the lines get the optimal height and depth so then no skips are inserted:

these are a few
lines of text

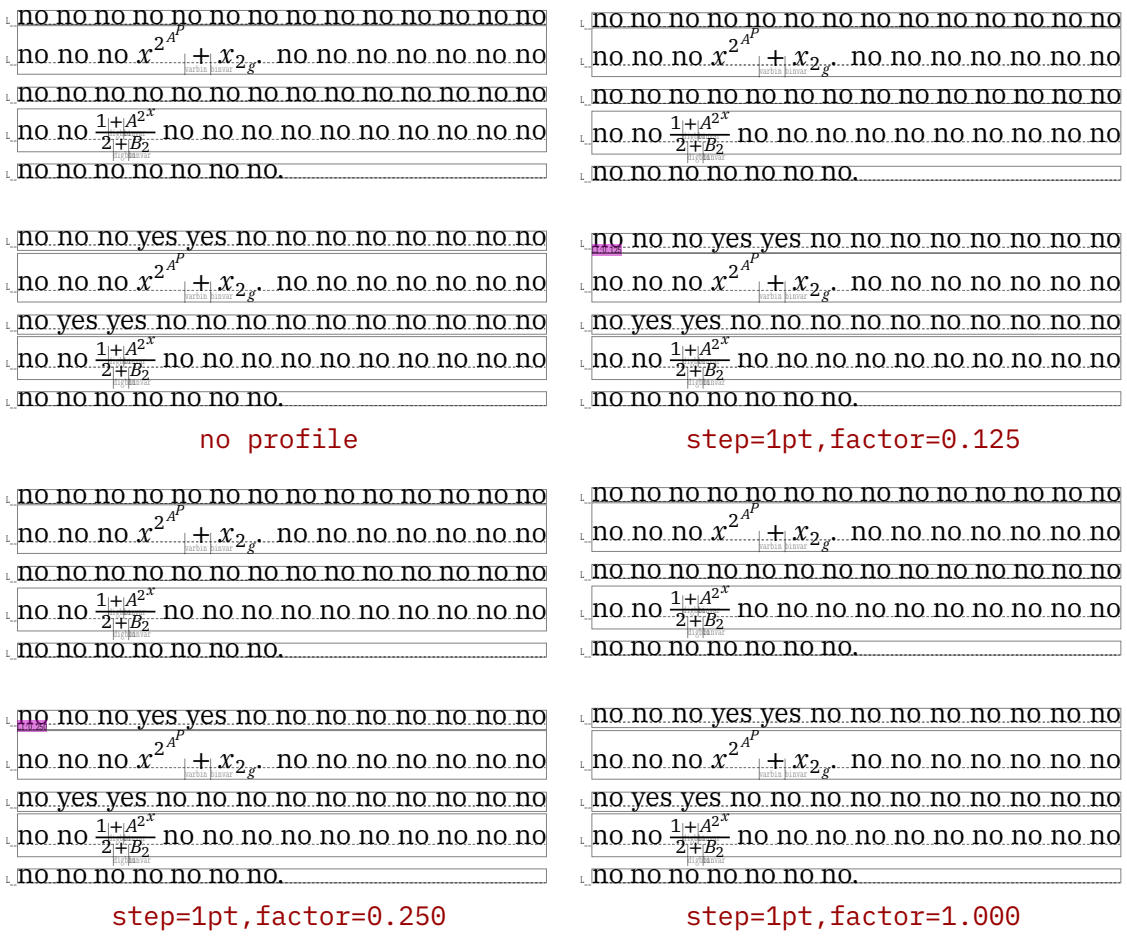


Figure 17.1

where lines have depths
or no real
height at all

When we increase the depth a little, for instance 1.2 times the normal strut depth, we see that some additional space, the `\lineskip` gets added:

these are a few
lines of text
where lines have depths
or no real
height at all

However, there is no real reason to do that here because the larger rules don't clash with text or other content. So this is what we get when we pass the `profile` option to `\setupalign`:

these are a few
lines of text
where lines have depths
or no real
height at all

Profiling works per paragraph, so when we add a `\par` in the middle we get this:

these are a few
lines of text
where lines have depths
or no real
height at all

But, we can actually setup the profiler to look back. Setting up the main (document) profiler happens with:

```
\setuplineprofile
  [factor=0.125,      % default
   paragraph=yes,     % default: no
   step=0.5\emwidth]  % default
```

but as with most ConTEXt mechanisms you can define your own profiler. The step tells what granularity to use when comparing positions in a line. The factor sets the threshold for the interline skip. We saw these two differ in the first example we gave.

17.3 Profiled math

We will give several examples of math profiling. In the examples we will switch font to Latin Modern, since the effect is more visible for that font. Most of our examples will be "real" (slightly modified) ones, but we start with a rather artificial example. Below we have two occurrences of a fraction. Note that the profiling only kicks in for the second one. The reason is that on the line above the first one we only have letters (x) with no depth, while in the second one, we have added one letter (g) that has depth.

xxx xxxx xxxx xx xxx xx xx xxx xxxx xxx xx xxx xxx xx
xx xxx x xxxxx xxxxx xxx xxxx xxxx xx xxx xx xx xxx
xxxx xxx xx xxx xxx xx xx xxx x xxxxx xxxxx $\frac{1}{2}$ xxx xxx
xx xx xxx xxxx xxxxxxx xxxx xxxx xxxxx xxx xxxx xxx
xxx xxx xxx xxx xxx xxxx xxxxxx xxxxxxx xxxgxxx xxx
xxx xxx xx x xxxxxxx xxx $\frac{1}{2}$ xxx xxxx xxxx xx xxx xx xx
xxx xxxx xxx xx xxx xxx xx xx xxx x xxxxx xxxxx.

no profile

xxx xxxx xxxx xx xxx xx xx xxx xxxx xxx xx xxx xxx xx
xx xxx x xxxxx xxxxx xxx xxxx xxxx xx xxx xx xx xxx
xxxx xxx xx xxx xxx xx xx xxx x xxxxx xxxxx $\frac{1}{2}$ xxx xxx
xx xx xxx xxxx xxxxxxx xxxx xxxx xxxxx xxx xxxx xxx
xxx xxx xxx xxx xxx xxxx xxxxxx xxxxxxx xxxgxxx xxx
xxx xxx xx x xxxxxxx xxx $\frac{1}{2}$ xxx xxxx xxxx xx xxx xx xx
xxx xxxx xxx xx xxx xxx xx xx xxx x xxxxx xxxxx.

step=1pt,factor=0.125

We next show a simple paragraph where the mechanism gets applied in three out of five line breaks.

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p > N_p^0$, while the convergence may be slow if $N_p = N_p^0$. As the following proposition reveals, at least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

no profile

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p > N_p^0$, while the convergence may be slow if $N_p = N_p^0$. As the following proposition reveals, at least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

step=1pt,factor=0.125

We have shown the lines and used the helper to show where the profiling is applied. We show the same example but without these helpers. After all, this is how we usually see it.

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. As the following proposition reveals, at least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

no profile

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. As the following proposition reveals, at least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

step=1pt,factor=0.125

If the paragraph is slightly reformulated, the profiling might change. Below we show an example where the subscript (p) on the fourth line gets too close to the superscript (0) on the last line.

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. At least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

no profile

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. At least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

step=1pt,factor=0.125

We can configure the amount of space that shall be added with the `factor` key.

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. At least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

step=1pt,factor=0.125

The results of Section 6.3 show that the same phenomenon is encountered when treating the norms of inverses: $\|B_n^{-1}\|_p$ converges to N_p very fast if $N_p \geq N_p^0$, while the convergence may be slow if $N_p = N_p^0$. At least for $p = 2$ the strict inequality $N_2 > N_2^0$ is the generic case.

step=1pt,factor=1

This shows that the sequence (f_n) is a Cauchy sequence in $L^1(\mathbb{R})$ and therefore it converges in norm to some $\sum_{k=1}^{n} \hat{f} \in L^1(\mathbb{R})$, by Thereom 2.8.1. On the other hand, by Theorem 2.8.2, there exists an increasing sequence of positive integers q_n such that $f_{q_n} \to \tilde{f}$ almost everywhere.

no profile

This shows that the sequence (f_n) is a Cauchy sequence in $L^1(\mathbb{R})$ and therefore it converges in norm to some $\sum_{k=1}^{n} \hat{f} \in L^1(\mathbb{R})$, by Thereom 2.8.1. On the other hand, by Theorem 2.8.2, there exists an increasing sequence of positive integers q_n such that $f_{q_n} \to \tilde{f}$ almost everywhere.

step=1pt,factor=0.125

17.4 Line spacing

When we enable the line profiler on a 300 page math course with plenty inline formulas, the number of 'corrections' varies a lot with the fonts. Some simple tests show that Latin Modern, Bonum and EBGaramond get quite some applied, while Lucida, Dejavu, Antykwa, Erewhon and Libertinus only see a few corrections. Pagella, Termes and StixTwo end up in the middle.

The trigger is not always text or math. The course material has quite some structure, like numbered descriptions. In ConTEXt we use plenty of struts to make sure that spacing is consistent and the keyword that starts a description therefore gets them. Normally that is not an issue but when the height of the next line exceeds the strut height we get a clash and line skip will be added. One can argue that the strut spoils the typesetting but in general it does more good than harm, at least in ConTEXt. It looks like the profiler is quite

capable of getting rid of the cases where it interferes (or more precisely: where it doesn't run into the next line).

The reason why we get a line skip added is simple: when the depth of the first line equals strut depth and the height of the second one equals strut height we're okay. When one of them is less we're also okay because TEX will adapt the baseline skip so that it compensated the difference. However, when the first line has strut depth (due to the present strut) and the second line more than strut height (resulting for instance from a formula) the lines are considered overflowing in each other and therefore interline skip gets added.

When we end up in this situation the profiler can bring down the line skip when it concludes that the strut is not running into the next line. However when the formula sits directly below the strut we cannot really determine what is right so then we just keep the skip. This situation occurs seldom. In many cases struts are optional so one can always disable them (locally).

As the mentioned test document uses Lucida as body font, in the three cases where we actually get a clash, one definitely relates to the strut: the overflow in the second line occurs close to the right margin and the strut in the first line sits at the left margin so we can get rid of the line skip, which leave us with only two cases. However, there is another observation, one that involves the baseline distance or line height.

In ConTEXt the ratio between the strut height and depth is 72:28 which works quite well for most fonts. If we look at Lucida shapes we see that the depth is normally small so we can actually decide to change that ratio. It is however not clear how that will influence decisions. Assuming more height will help with for instance formulas that have superscripts, but an inline integral with subscript might suffer. For other fonts, like EBGaramond, that have some extremely deep shapes changing rations won't help anyway. We win here and loose there.

We will look into a few fonts to get a better impression how all this relates. We will use 10pt sizes. When we compare Lucida, Latin Modern, Bonum and Pagella we notice that we start out with design sizes that are quite different.

lucida

strut ht	:	10.67896pt
strut dp	:	4.15291pt
ex	:	5.29709pt

modern

strut ht	:	8.68419pt
strut dp	:	3.37718pt
ex	:	4.30763pt

bonum

strut ht	:	9.77223pt
strut dp	:	3.80031pt
ex	:	4.84734pt

pagella

strut ht	:	9.44986pt
strut dp	:	3.67493pt
ex	:	4.68742pt

This is why we always use ratios (the 0.72 and 0.28) as well as abstract dimensions like `ex` and `em` so that we adapt to what the font provides. Because the default (total) line height is set to `2.8ex` we get larger values in for instance Lucida.

In the next tables we use three samples, with `text-001` being:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
()[]/\{\}\|
.,!?@\#\$\%^&*_-+=
```

The two math samples are:

```
$\int\sum\sqrt{}$
```

and

```
x_2^2
```

For text the ratios are not that far off the defaults, but for math they start to differ and the distance becomes larger. For Lucida we get:

text-001		math-001		math-002	
0.79	7.8257pt	0.81	9.81741pt	0.92	7.40593pt
0.21	2.04887pt	0.19	2.24377pt	0.08	0.62965pt

Modern gives:

text-001		math-001		math-002	
0.75	7.49588pt	0.75	9.16898pt	0.84	7.43591pt
0.25	2.49863pt	0.25	3.05333pt	0.16	1.37924pt

And bonum moves in the other direction:

text-001		math-001		math-002	
0.77	7.90565pt	0.77	10.16666pt	0.82	7.21603pt
0.23	2.40868pt	0.23	3.11829pt	0.18	1.54915pt

Pagella also differs:

text-001		math-001		math-002	
0.73	7.49588pt	0.79	11.09692pt	0.85	6.88622pt
0.27	2.82845pt	0.21	2.95837pt	0.15	1.24931pt

Changing the ratios for the sake of math makes not that much sense because profiling depends a lot on what math ends up inline. When we looked around a bit for realistic examples we got the impression that seeing some clash (read: getting uneven line spacing) might be a reason why small formulas eventually end up as display. Without mentioning names, we noticed that a reprint of a book actually got reformatted and when we looked for the slashing formulas in the original they had become display instead. With proper profiling there is no need for that. On top of that one can argue that some inline rendering can be done better anyway, like using skewed fraction instead of ruled ones. Can we predict that math with many superscripts goes well with a font with relatively high shapes? As soon as some parenthesis are used we get depth anyway and how likely is it that math without these is used inline? It also depends on the amount of math: two lines with superscripted math will drive TEX to use line skip so we need to profile anyway. For that reason we will not adapt the ratios, just like we keep the default line spacing. There are of course fonts with extreme heights (like the Computer Modern Dunhill variant) but no one will use those artistic variants in a math document.

If you want to go fancy and distinctive, Antykwa is a good choice and that one actually scores pretty good with the defaults!

antykwa	
strut ht	: 9.47pt
strut dp	: 3.68277pt
ex	: 4.69742pt

text-001		**math-001**		**math-002**	
0.75	7.69577pt	0.76	9.59224pt	0.87	7.0961pt
0.25	2.49863pt	0.24	3.05562pt	0.13	1.04942pt

17.5 Conclusion

So, should we enable profiling on mixed text-math documents or not? One possible reason for not doing it is that it adds overhead, but in practice it's not that much compared to processing the rest. It is no problem to find complaints on the internet about LuaTEX performing worse than its ancestors so if you're in that category: don't use profiling because it sets you back a few percent runtime. However, when you're a demanding ConTEXt user who mixes in a lot of math, you might give it a try. It will intercept a couple of cases where struts (assuming structure is used) trigger a line skip, and it might also catch a couple of cases where TEX found lines getting too close. Tweaking the `factor` and `step` can actually be fun. Because it does influence page breaks it is not something to be applied last minute. And, talking performance, this kind of vertical optimization comes cheaper than horizontal optimization using expansion (hz) and protrusion.

18 Pushing the envelope

Here I describe the results of some exploration and experiments by Mikael Sundqvist and me. We got sidetracked from intersections, arcs and drawing functions when we noticed some artifacts with envelopes. But what are envelopes actually? Let us start with a simple path:

```
\startMPinclusions
    path TestPath ; TestPath := fullcircle xyscaled (10cm,1cm)
\stopMPinclusions
```

When we draw this with a circular pen we get this:

```
\startMPcode
    draw TestPath withpen pencircle scaled 2mm withcolor darkred ;
\stopMPcode
```

Filling gives:

```
\startMPcode
    fill TestPath withpen pencircle scaled 2mm withcolor darkred  ;
\stopMPcode
```

When a `pencircle` is used MetaPost delegates the work to the backend because PostScript has a circular pen, otherwise it has to calculate the to-be-filled shape itself. The backend has to do some path juggling in the case of pdf because there a pen transform is different from PostScript.

```
\startMPcode
    draw TestPath withpen pensquare scaled 2mm withcolor darkblue ;
\stopMPcode
```

Here we draw the shape with a square pen while filling gives:

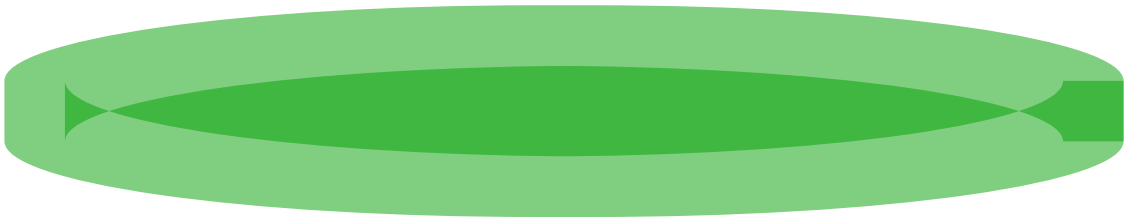
```
\startMPcode
    fill TestPath withpen pensquare scaled 2mm withcolor darkblue ;
\stopMPcode
```

In most cases this works out well but there are some hidden issues. These get exposed when we use a transparency:

```
\startMPcode
    fill TestPath withpen pensquare scaled 2mm withcolor darkgreen
        withtransparency (1,.5) ;
\stopMPcode
```

It are these artifacts that we will explore a little. For that we will render quite some graphics. We could show numerous more examples but when you are a ConTEXt user you will be able to make plenty yourself by looking at these examples.

```
\startMPcode
    fill fullcircle xyscaled (.8TextWidth,2cm)
        withpen pensquare scaled 8mm
        withcolor darkgreen
        withtransparency (1,.5) ;
\stopMPcode
```



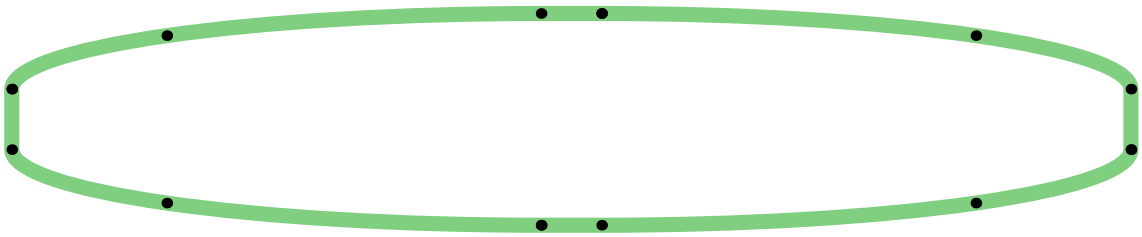
When we were playing with the `envelope` primitive we noticed these artifacts and we spent quite some time looking at the code to see where it comes from and if we could prevent this. It was then that we realized that the fill actually also uses these envelopes but that it gets delayed till the shapes are flushed to the backend. That meant that we could use fills with transparencies as simple test cases.

The first thing to get rid of is the weird blob at the right end of the fill in this example. Not really understanding all what went on, we explored all kind of shapes and temporarily disabled some of the code in the MetaPost library to see where it crept in. We decided that touching the code to get rid of for instance rounding issues or potential direction related side effects made no sense. In the end the solution was simple:

```
\startMPcode
    pen p ; p := makepen(unitsquare rotated eps) ;
    fill fullcircle xyscaled (.8TextWidth,2cm)
        withpen p scaled 8mm
        withcolor darkgreen
        withtransparency (1,.5) ;
\stopMPcode
```

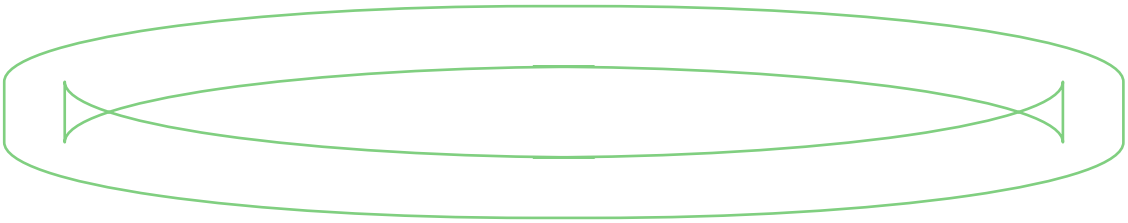
We show what the `envelope` primitive gives us:

```
\startMPcode
    pen p ; p := makepen(unitsquare rotated eps) ;
    path e ; e :=
        envelope (p scaled 8mm)
    of
        (fullcircle xyscaled (.8TextWidth,2cm))
    ;
    draw e
        withpen pencircle scaled 2mm
        withcolor darkgreen
        withtransparency (1,.5) ;
    drawpoints e ;
\stopMPcode
```



This looks okay compared to previous the examples but we have only a simple path here, while the fill actually has two:

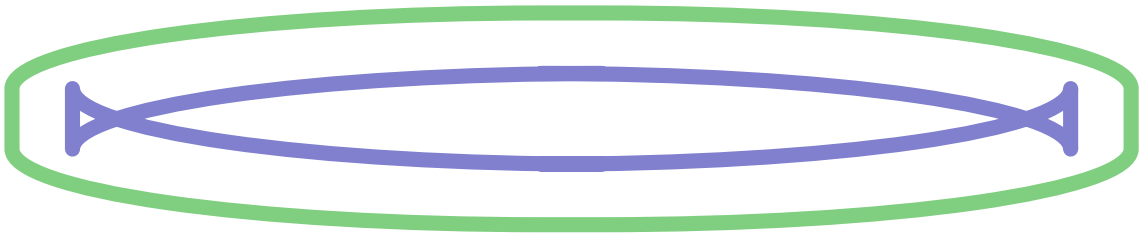
```
\startMPcode
    pen p ; p := makepen(unitsquare rotated eps) ;
    enfill fullcircle xyscaled (.8TextWidth,2cm)
        withpen p scaled 8mm
        withcolor darkgreen
        withtransparency (1,.5) ;
\stopMPcode
```



So how do we get that inner shape? Once you know what a fill actually outputs to the backend it is easy! There are two envelopes: the normal one and one made from the reverse path (or in internal MetaPost speak: htap). In the previous example the `enfill` treats the path as a fill but will draw the envelopes instead. As with `eofill`, `eoclip` and path accumulators this is a MetaFun backend related feature but we introduced `enfill` as a new one.

```
\startMPcode
    pen p ; p := makepen(unitsquare rotated eps) ;
    draw
        envelope (p scaled 8mm) of
        (fullcircle xyscaled (.8TextWidth,2cm))
        withpen pencircle scaled 2mm
```

```
        withcolor darkgreen
        withtransparency (1,.5) ;
    draw
        envelope (p scaled 8mm) of
        reverse (fullcircle xyscaled (.8TextWidth,2cm))
        withpen pencircle scaled 2mm
        withcolor darkblue
        withtransparency (1,.5) ;
\stopMPcode
```



We're now ready for the real deal but keep in mind that what we show here is the result of stepwise growing insight combined with adding some features to the engine that not only makes it possible to illustrate this but also might prove to be useful. The used primitives will be explained later, for now we just stick to the results.

Figure 18.1 shows a circle filled (or enveloped) with pens made from `fullcircle`, `fulldiamond`, `fulltriangle` and `fullsquare`. The paths that we use for the pens are also shown. The outcome can be puzzling but after going over the code (in the engine) and trying to reason the logic it becomes clear that the unexpected is mostly due to the fact that there is no other way to draw the path (read: meet the criteria).

When looking closely at the results (adding labels to the points and zooming in) one will notice more side effects. Because we rotate over `eps` to get rid of the weird end situation we can end up with more points than we like and these are so close to each other that one doesn't notice them. For this we can apply the scrutinizer:

```
e := e scrutinized 0.01 ;
```

When that is done we can wonder if a simplified (inner) path is possible. I tried a few solutions using the Lua interface while Mikael (as mathematician) followed the more scientific approach but the results largely depend on the pens and shapes.

Actually when doing all that we used a more complex pen in several variants. This is shown in figure 18.2. Notice the dashed lines here. When a pen is defined there is some checking going on. One is that circular pens get no treatment at all and just pass through the system. Basically any single point cycle is considered as elliptical anyway. Then the path turned into a so called 'convex' path. It also showed us the real pen being used. When out of curiosity I commented that bit of code I noticed that we could achieve interesting results. The result is that we now have a `convexed` primitive. After all the code was there so it took only a few lines to add this primitive. In figure 18.3 you can see the result of a unconvexed pen.

We can also calculate envelopes of non-cyclic paths which is demonstrated in figure 18.4 and figure 18.5. There is however some trickery involved. Just to make this easier the MetaFun macro package has a type starring macro that makes such a star:

```
path p ; p := starring(-1/2) rotated eps ;
```

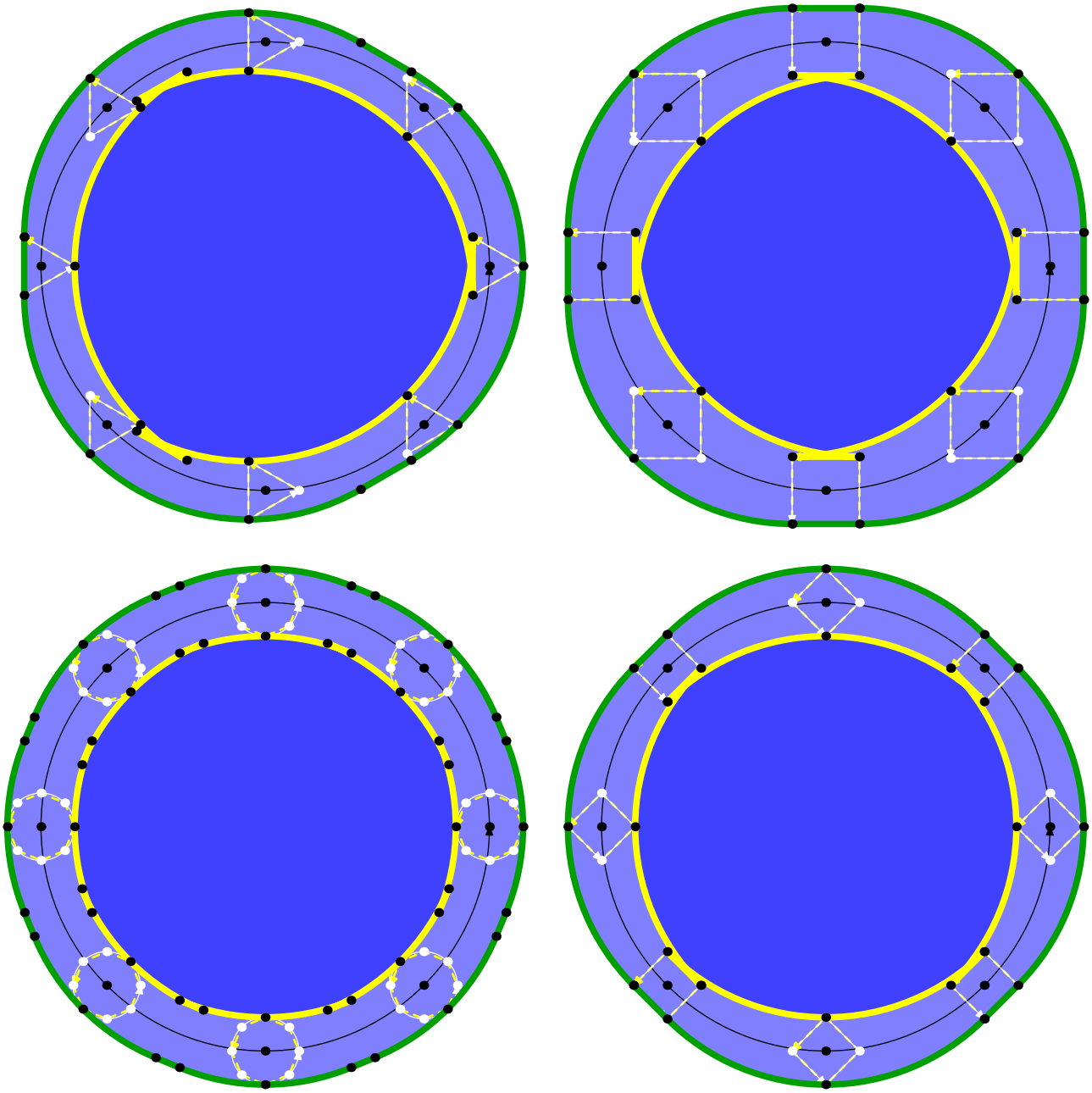


Figure 18.1 Using four different relatively large pens on a circle.

This star can become a pen:

```
pen somepen ; somepen := makepen (pp) ;
```

And as mentioned pens get convexed by default. Even worse, whenever we transform a pen it gets convexed again. When we fill a shape the pen gets attached to that shape and the backend will do the enveloping. The easiest way to consistently avoid convexing was to introduce a new pen type.

```
nep somepen ; somepen := makenep (pp) ;
```

The somewhat weird short `nep` perfectly fits the bill as in Dutch it means fake. A pen defined this way stays unconvexed. Actually there is another property where pens differ from regular paths: they are double linked. In original MetaPost that back (prev) link uses a field in a knot record that is not used by pen paths. The path that gets pencilled also abuses one of knot fields for keeping track of the offset that a point has relative to the current point in the pen. It was good moment to also make regular paths double linked lists.

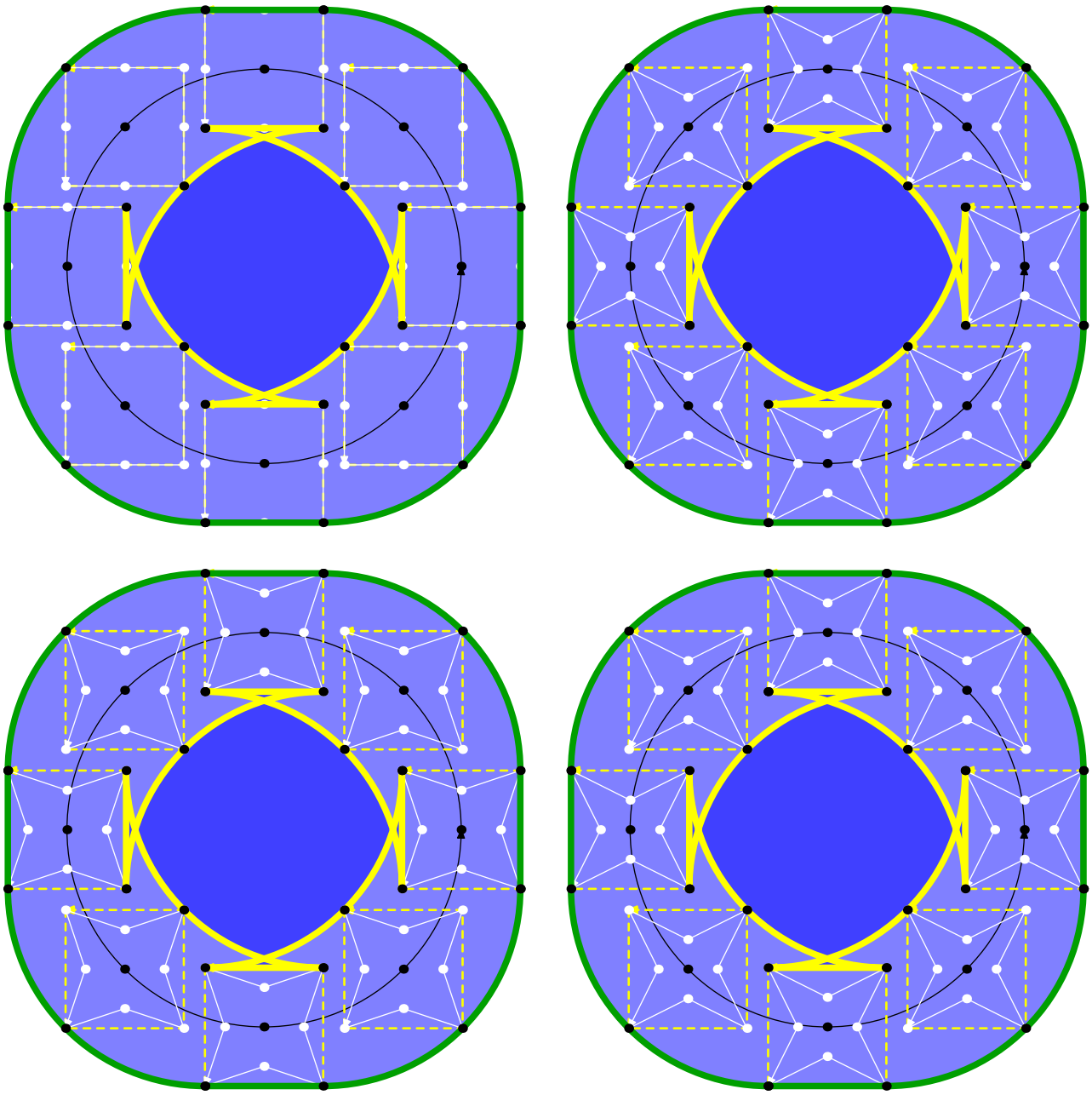


Figure 18.2

That comes at the cost of an extra pointer in the knot record but we could also save some space by using smaller slots for other fields. Memory is not our biggest worry anyway.[23]. Double linking meant that there was no need for doing that when making pens.[24]

We can apply the convexed primitive to the inner envelope which is demonstrated in figure 18.6 and figure 18.7. Of course it is debatable how useful this is but as with all these MetaPost shapes, it has some charm.

[23] Of course adding code is but when looking in more detail at the code involved it was actually possible to simplify the code a bit so there we gained

[24] It makes it possible to get points relative to the current point in iterators over paths that we introduced a while ago, which makes for high performance path manipulators.

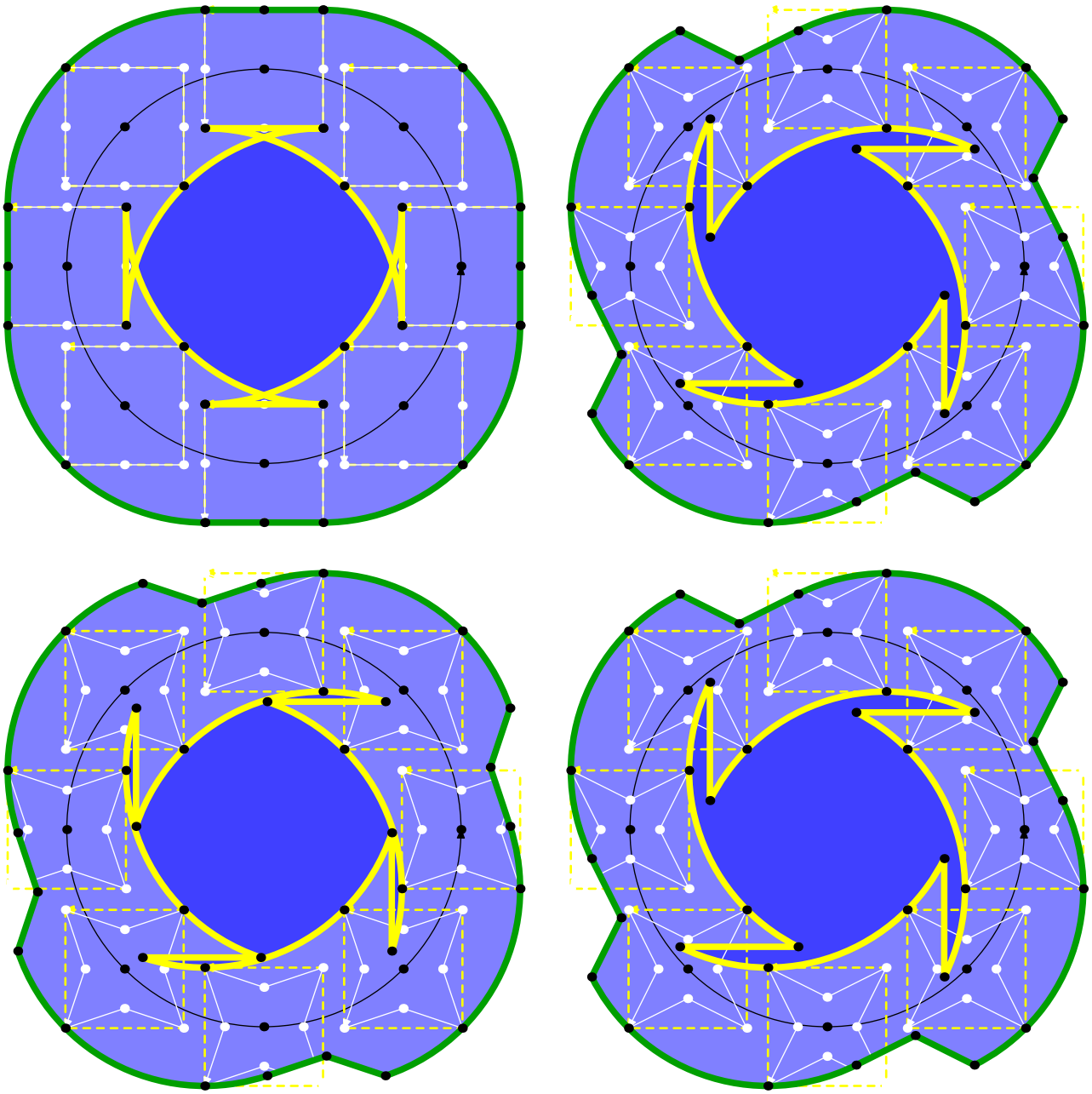


Figure 18.3

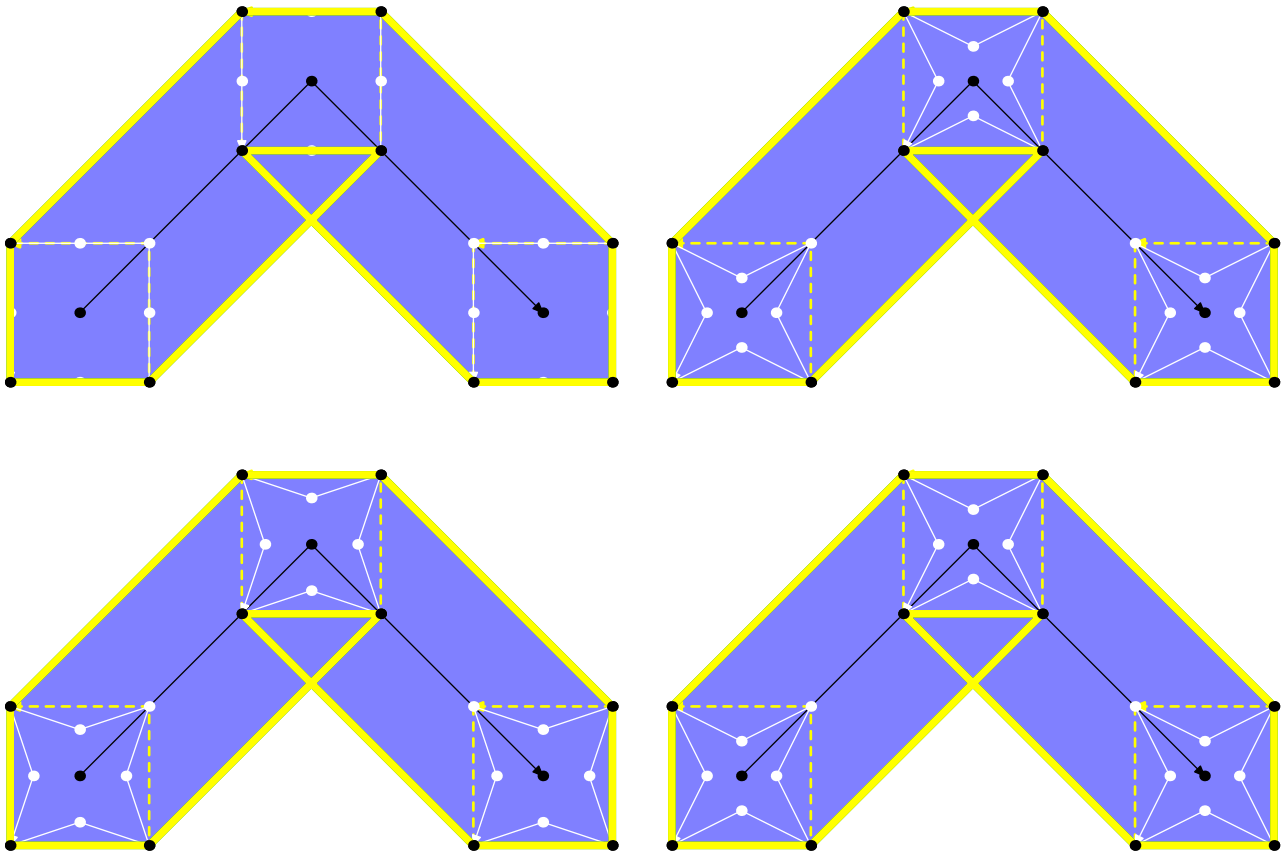


Figure 18.4

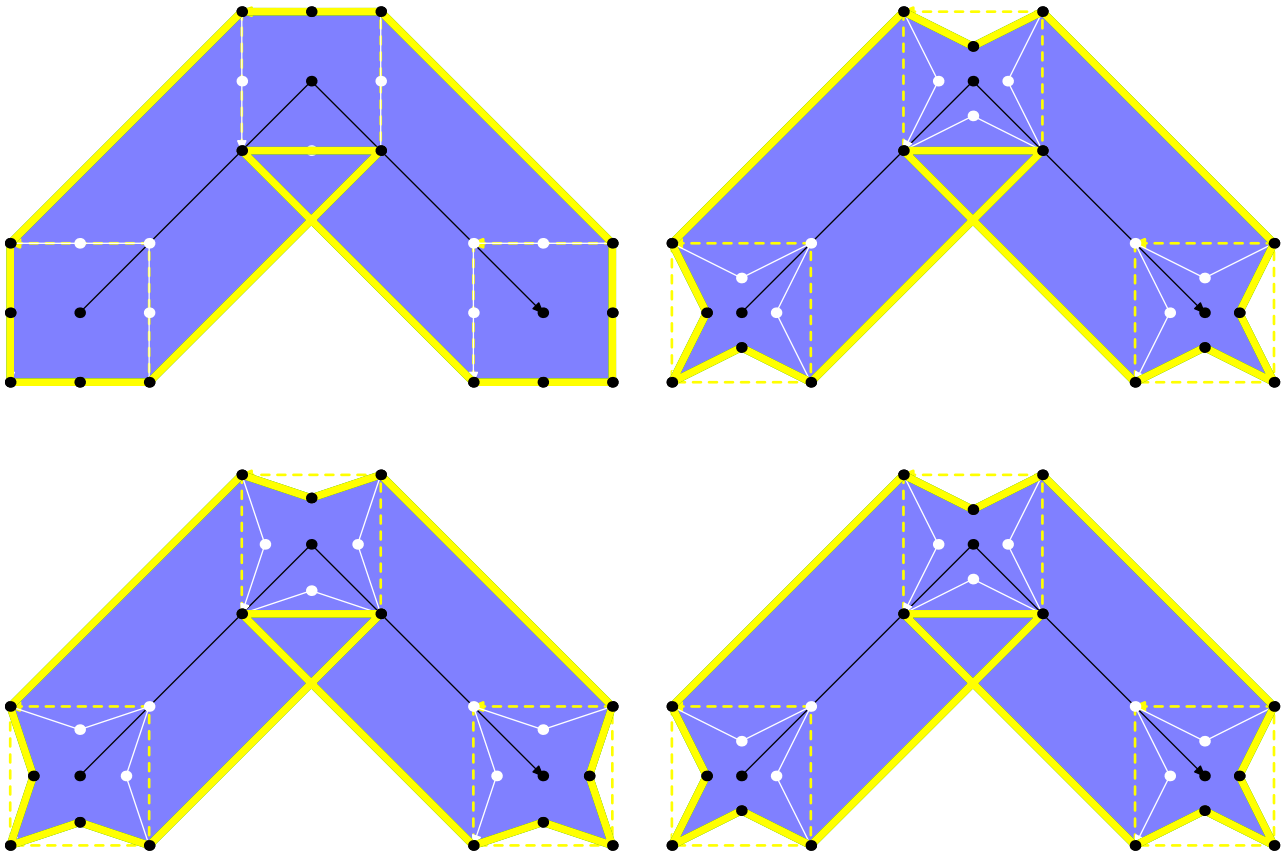


Figure 18.5

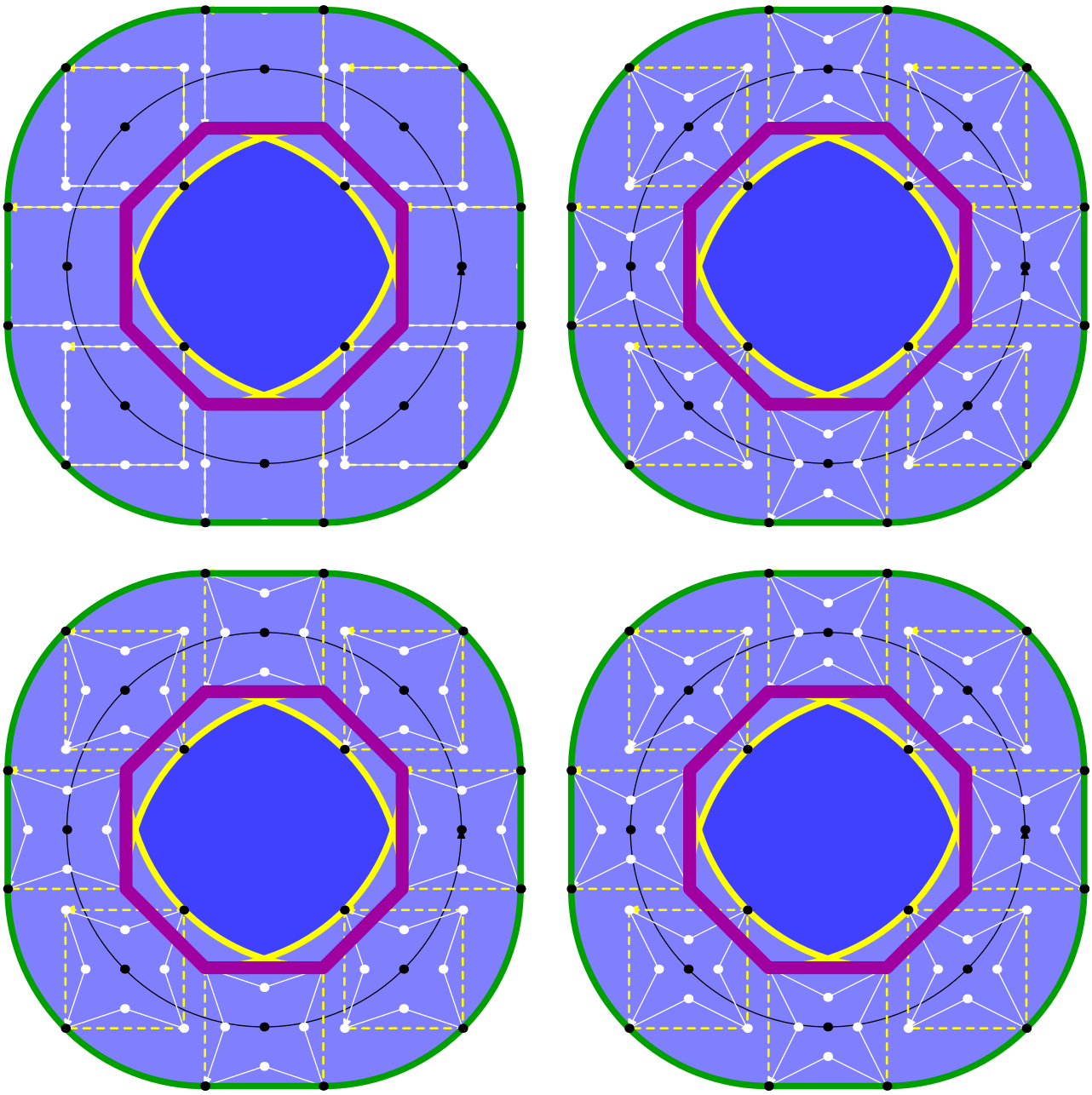


Figure 18.6

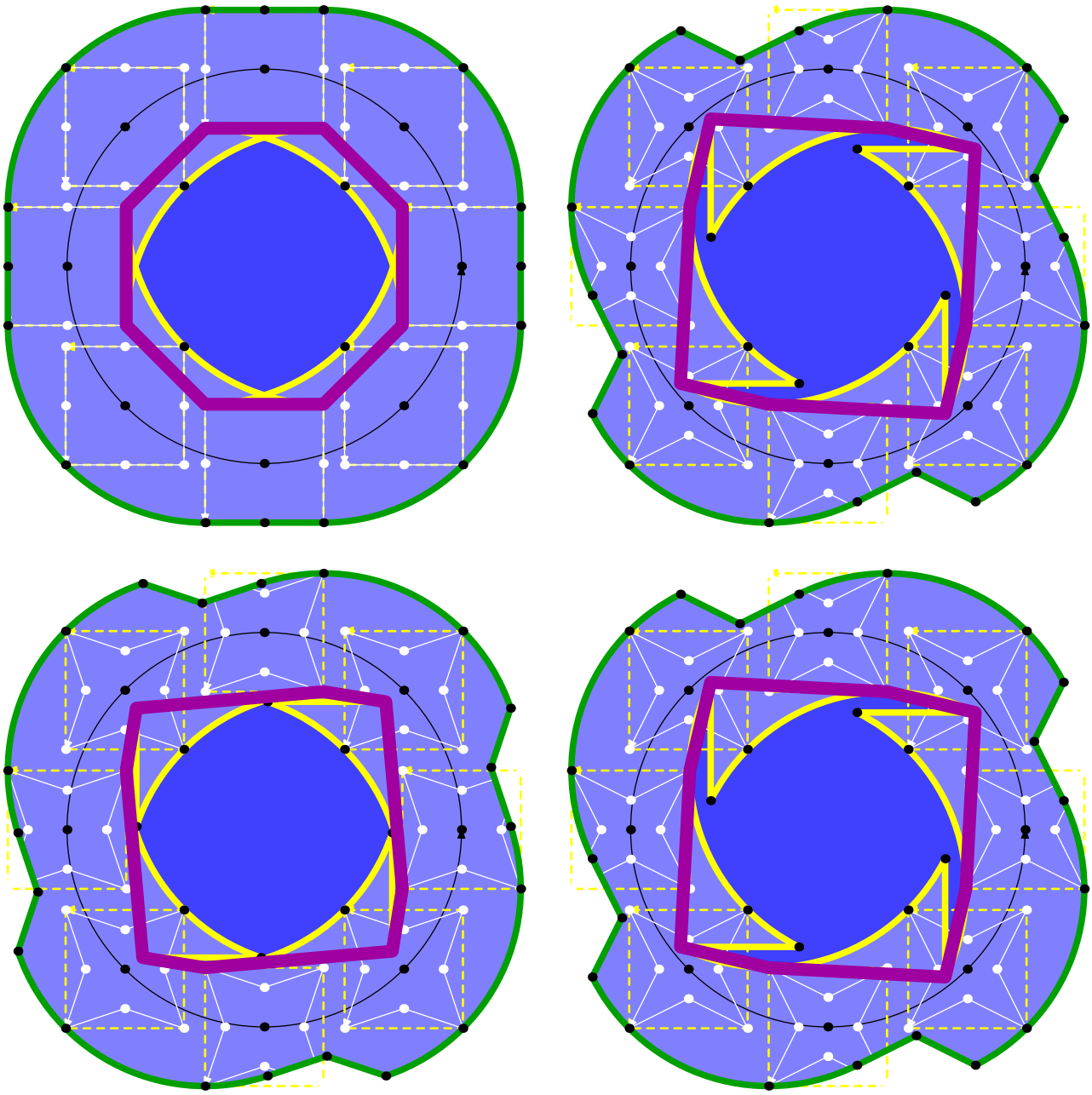
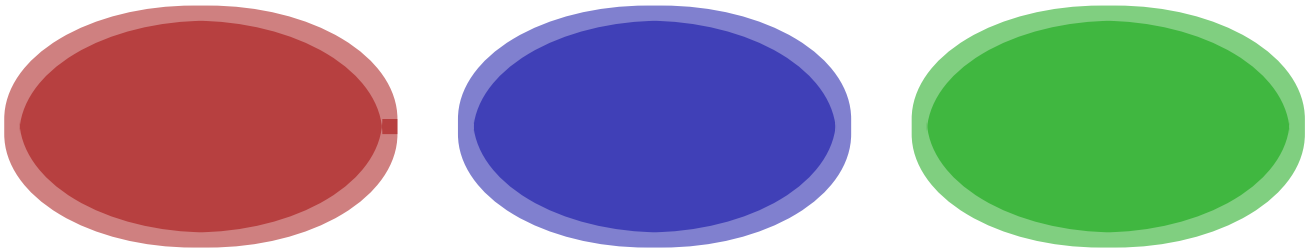


Figure 18.7

To what extend does all this influence the output? As long as we don't use transparencies we're quite okay unless we use a pen size that introduces the more extreme overshoots. If you think these phenomena only relate to MetaPost output, you're wrong. Over the past decades I've seen various fonts that exhibit the same small spikes and other artifacts btu as we often see the shapes at small sizes it goes unnoticed. A particular sensitive areas is variable fonts where, when the ranges on which the various dimensions operate are too liberal, you can also get these effects. After all, glyphs are filled shapes. To that you can also add the fact that they are single (connected) paths drawn with `eofill`.

The final format a graphics ends up in can be pdf. Take the following three shapes and watch the subtle side effect of rotating either the to be drawn shape or the pen.

```
\startMPcode
    fill fullcircle xyscaled (5cm,3cm)
        withpen makepen(fullsquare) scaled 2mm
        withcolor darkred
        withtransparency (1,.5) ;
    fill fullcircle xyscaled (5cm,3cm)
        shifted (6cm,0)
        withpen makepen(fullsquare rotated eps) scaled 2mm
        withcolor darkblue
        withtransparency (1,.5) ;
    fill fullcircle rotated eps xyscaled (5cm,3cm)
        shifted (12cm,0)
        withpen makepen(fullsquare) scaled 2mm
        withcolor darkgreen
        withtransparency (1,.5) ;
\stopMPcode
```



This produces four filled paths in the pdf file, a normal and a reverse path per shape. I show the whole output because you can see how some points of the 'inside' curve are sort of duplicated: they have the same coordinates but can have different control points.

	inner	outer
left	25=26 31=32 35=36 39=40 43=25	
middle		49=50=51 54=55 58=59 62=63
right	107=108=109 112=113 116=117 120=121	89=105

Here is the output. Each combination is between bound by the transparency operators `/Tr1` and `/Tr0` and has different colors.

```
% mps graphic 1: begin
q
/Tr1 gs
0.625 0 0 rg 0.625 0 0 RG
```

```
10 M
1 j
 45.354315 2.83464 m
 45.354315 14.111559 40.874577 24.926605 32.900591 32.900591 c
 24.926605 40.874577 14.111559 45.354315 2.83464 45.354315 c
  2.83464 45.354315 -2.83464 45.354315 -2.83464 45.354315 c
 -2.83464 45.354315 l
-14.111559 45.354315 -24.926605 40.874577 -32.900591 32.900591 c
-40.874577 24.926605 -45.354315 14.111559 -45.354315 2.83464 c
-45.354315 2.83464 -45.354315 -2.83464 -45.354315 -2.83464 c
-45.354315 -2.83464 l
-45.354315 -14.111559 -40.874577 -24.926605 -32.900591 -32.900591 c
-24.926605 -40.874577 -14.111559 -45.354315 -2.83464 -45.354315 c
-2.83464 -45.354315 2.83464 -45.354315 2.83464 -45.354315 c
2.83464 -45.354315 l
14.111559 -45.354315 24.926605 -40.874577 32.900591 -32.900591 c
40.874577 -24.926605 45.354315 -14.111559 45.354315 -2.83464 c
45.354315 -2.83464 45.354315 2.83464 45.354315 2.83464 c
45.354315 2.83464 l
h f
39.685035 -2.83464 m
39.685035 -2.83464 45.354315 -2.83464 45.354315 -2.83464 c
45.354315 -2.83464 45.354315 2.83464 45.354315 2.83464 c
45.354315 2.83464 39.685035 2.83464 39.685035 2.83464 c
39.685035 -8.442279 35.205297 -19.257325 27.231311 -27.231311 c
19.257325 -35.205297 8.442279 -39.685035 -2.83464 -39.685035 c
-2.83464 -39.685035 l
-2.83464 -39.685035 2.83464 -39.685035 2.83464 -39.685035 c
-8.442279 -39.685035 -19.257325 -35.205297 -27.231311 -27.231311 c
-35.205297 -19.257325 -39.685035 -8.442279 -39.685035 2.83464 c
-39.685035 2.83464 l
-39.685035 2.83464 -39.685035 -2.83464 -39.685035 -2.83464 c
-39.685035 8.442279 -35.205297 19.257325 -27.231311 27.231311 c
-19.257325 35.205297 -8.442279 39.685035 2.83464 39.685035 c
2.83464 39.685035 l
2.83464 39.685035 -2.83464 39.685035 -2.83464 39.685035 c
8.442279 39.685035 19.257325 35.205297 27.231311 27.231311 c
35.205297 19.257325 39.685035 8.442279 39.685035 -2.83464 c
39.685035 -2.83464 l
h f
/Tr0 gs
0 g 0 G
/Tr1 gs
0 0 0.625 rg 0 0 0.625 RG
158.740139 -2.834616 m
158.740139 -2.834252 l
158.740139 -2.834252 158.740091 2.835028 158.740091 2.835028 c
158.739994 14.111816 154.260267 24.926715 146.286366 32.900615 c
138.31238 40.874601 127.497335 45.354339 116.220416 45.354339 c
```

```
116.220052 45.354339 l
116.220052 45.354339 110.550772 45.354291 110.550772 45.354291 c
99.273984 45.354194 88.459085 40.874467 80.485185 32.900566 c
72.511199 24.92658 68.031461 14.111535 68.031461 2.834616 c
68.031461 2.834252 l
68.031461 2.834252 68.031509 -2.835028 68.031509 -2.835028 c
68.031606 -14.111816 72.511333 -24.926715 80.485234 -32.900615 c
88.45922 -40.874601 99.274265 -45.354339 110.551184 -45.354339 c
110.551548 -45.354339 l
110.551548 -45.354339 116.220828 -45.354291 116.220828 -45.354291 c
127.497616 -45.354194 138.312515 -40.874467 146.286415 -32.900566 c
154.260401 -24.92658 158.740139 -14.111535 158.740139 -2.834616 c
h f
153.070811 2.834616 m
153.070811 -8.442303 148.591072 -19.257349 140.617086 -27.231335 c
132.643186 -35.205235 121.828288 -39.684963 110.5515 -39.685059 c
110.5515 -39.685059 116.22078 -39.685011 116.22078 -39.685011 c
116.220416 -39.685011 l
104.943497 -39.685011 94.128451 -35.205272 86.154465 -27.231286 c
78.180565 -19.257386 73.700837 -8.442488 73.700741 2.8343 c
73.700741 2.8343 73.700789 -2.83498 73.700789 -2.83498 c
73.700789 -2.834616 l
73.700789 8.442303 78.180528 19.257349 86.154514 27.231335 c
94.128414 35.205235 104.943312 39.684963 116.2201 39.685059 c
116.2201 39.685059 110.55082 39.685011 110.55082 39.685011 c
110.551184 39.685011 l
121.828103 39.685011 132.643149 35.205272 140.617135 27.231286 c
148.591035 19.257386 153.070763 8.442488 153.070859 -2.8343 c
153.070859 -2.8343 153.070811 2.83498 153.070811 2.83498 c
153.070811 2.834616 l
h f
/Tr0 gs
0 g 0 G
/Tr1 gs
0 0.625 0 rg 0 0.625 0 RG
272.125915 2.835004 m
272.125819 14.111923 267.645988 24.92693 259.671934 32.900848 c
251.697965 40.87468 240.883028 45.354315 229.606241 45.354315 c
229.606241 45.354315 223.936961 45.354315 223.936961 45.354315 c
223.936596 45.354315 l
212.659677 45.354219 201.84467 40.874388 193.870752 32.900334 c
185.89692 24.926365 181.417285 14.111428 181.417285 2.834641 c
181.417285 2.834641 181.417285 -2.834639 181.417285 -2.834639 c
181.417285 -2.835004 l
181.417381 -14.111923 185.897212 -24.92693 193.871266 -32.900848 c
201.845235 -40.87468 212.660172 -45.354315 223.936959 -45.354315 c
223.936959 -45.354315 229.606239 -45.354315 229.606239 -45.354315 c
229.606604 -45.354315 l
240.883523 -45.354219 251.69853 -40.874388 259.672448 -32.900334 c
```

```
267.64628 -24.926365 272.125915 -14.111428 272.125915 -2.834641 c
272.125915 -2.834641 272.125915 2.834639 272.125915 2.834639 c
272.125915 2.835004 l
h f
266.456635 -2.834276 m
266.456635 -2.834641 l
266.456635 -2.834641 266.456635 2.834639 266.456635 2.834639 c
266.456635 -8.442148 261.977 -19.257085 254.003168 -27.231054 c
246.02925 -35.205108 235.214243 -39.684939 223.937324 -39.685035 c
223.936959 -39.685035 l
223.936959 -39.685035 229.606239 -39.685035 229.606239 -39.685035 c
218.329452 -39.685035 207.514515 -35.2054 199.540546 -27.231568 c
191.566492 -19.25765 187.086661 -8.442643 187.086565 2.834276 c
187.086565 2.834641 l
187.086565 2.834641 187.086565 -2.834639 187.086565 -2.834639 c
187.086565 8.442148 191.5662 19.257085 199.540032 27.231054 c
207.51395 35.205108 218.328957 39.684939 229.605876 39.685035 c
229.606241 39.685035 l
229.606241 39.685035 223.936961 39.685035 223.936961 39.685035 c
235.213748 39.685035 246.028685 35.2054 254.002654 27.231568 c
261.976708 19.25765 266.456539 8.442643 266.456635 -2.834276 c
h f
/Tr0 gs
0 g 0 G
Q
% mps graphic 1: end
```

The duplicates differ per variant and as they are effective `lineto` combine with `curveto` we can consider removing the `lineto`'s. This can either be done in the backend or we can decide to do that in the MetaPost library during the export.[25]

The `tracingspecs` flag can help us to see what happens deep down when

envelopes are made. It will show the intermediate path on the console.

```
tracingspecs := 1;
path e ; e := envelope (makepen(fullsquare) scaled 2mm) of (fullcircle scaled 3cm)
                                                                              ;
show(e);
```

The intermediate path is reported as:

```
   % beginning with      offset ( 2.83464, 2.83464)
   ( 42.51968, 0       ) .. controls ( 42.51968, 11.27742) and ( 38.03908,
                                                                    22.09160)
.. ( 30.06534, 30.06534) .. controls ( 22.09160, 38.03908) and ( 11.27742,
                                                                    42.51968)
   % counterclockwise to offset (-2.83464, 2.83464)
```

[25] In the end I settled on introducing a move tolerance in addition to the bend tolerance that we already have in the export. The default value of 0.001999 removes 14 lines from the above pdf code.

```
.. (  0,       42.51968) .. controls (  0,       42.51968) and (  0,
                                                                  42.51968)
.. (  0,       42.51968) .. controls (-11.27742, 42.51968) and (-22.09160,
                                                                  38.03908)
.. (-30.06534, 30.06534) ..controls (-38.03908, 22.09160) and (-42.51968, 11.27742)
   % counterclockwise to offset (-2.83464,-2.83464)
.. (-42.51968,  0      ) .. controls (-42.51968,  0      ) and (-42.51968,  0
                                                                        )
.. (-42.51968,  0      ) .. controls (-42.51968,-11.27742) and
                                                       (-38.03908,-22.09160)
.. (-30.06534,-30.06534) .. controls (-22.0916, -38.03908) and
                                                       (-11.27742,-42.51968)
   % counterclockwise to offset ( 2.83464,-2.83464)
.. (  0,      -42.51968) .. controls (  0,      -42.51968) and (  0,
                                                                 -42.51968)
.. (  0,      -42.51968) .. controls ( 11.27742,-42.51968) and ( 22.0916,
                                                                 -38.03908)
.. ( 30.06534,-30.06534) .. controls ( 38.03908,-22.09160) and (
                                                    42.51968,-11.27742)
   % counterclockwise to offset ( 2.83464, 2.83464)
.. ( 42.51968,  0      ) .. controls ( 42.51968,  0      ) and ( 42.51968,  0
                                                                        )
.. ( 42.51968,  0      )
&  cycle
```

The result becomes:

```
   ( 45.35432,  2.83464) .. controls ( 45.35432, 14.11206) and ( 40.87372,
                                                                  24.92624)
.. ( 32.89998, 32.89998) .. controls ( 24.92624, 40.87372) and ( 14.11206,
                                                                  45.35432)
.. (  2.83464, 45.35432) .. controls (  2.83464, 45.35432) and ( -2.83464,
                                                                  45.35432)
.. ( -2.83464, 45.35432) .. controls ( -2.83464, 45.35432) and ( -2.83464,
                                                                  45.35432)
.. ( -2.83464, 45.35432) .. controls (-14.11206, 45.35432) and (-24.92624,
                                                                  40.87372)
.. (-32.89998, 32.89998) .. controls (-40.87372, 24.92624) and (-45.35432,
                                                                  14.11206)
.. (-45.35432,  2.83464) .. controls (-45.35432,  2.83464) and (-45.35432,
                                                                  -2.83464)
.. (-45.35432, -2.83464) .. controls (-45.35432, -2.83464) and (-45.35432,
                                                                  -2.83464)
.. (-45.35432, -2.83464) .. controls (-45.35432,-14.11206) and
                                                       (-40.87372,-24.92624)
.. (-32.89998,-32.89998) .. controls (-24.92624,-40.87372) and
                                                       (-14.11206,-45.35432)
.. ( -2.83464,-45.35432) .. controls ( -2.83464,-45.35432) and (
                                                     2.83464,-45.35432)
```

```
.. (  2.83464,-45.35432) .. controls (  2.83464,-45.35432) and (
                                                        2.83464,-45.35432)
.. (  2.83464,-45.35432) .. controls ( 14.11206,-45.35432) and (
                                                       24.92624,-40.87372)
.. ( 32.89998,-32.89998) .. controls ( 40.87372,-24.92624) and (
                                                       45.35432,-14.11206)
.. ( 45.35432, -2.83464) .. controls ( 45.35432, -2.83464) and ( 45.35432,
                                                            2.83464)
.. ( 45.35432,  2.83464) .. controls ( 45.35432,  2.83464) and ( 45.35432,
                                                            2.83464)
.. cycle
```

Numerous experiments by Mikael and me lead to the conclusion that both stages can introduce the duplicate points and that any messing with that during envelop generation time has negative side effects. However, when we export the path we can definitely get rid of them. They are harmless but we're talking quality control here and TEX and MetaPost is all about quality!

As usual, playing with mechanisms like this gets one wondering about similar cases, for instance variants of dashing.

```
\startMPcode
vardef dashing (expr pth, shp, stp) =
    for i within arcpointlist stp of pth :
        shp
            rotated angle(pathdirection)
            shifted pathpoint
        &&
    endfor nocycle
enddef ;

path parrA ; parrA :=
    (0,0) -- (0,-1) -- (2,-1) -- (2,-2) -- (4,0) -- (2,2) -- (2,1) -- (0,1) --
                                                                        (0,0)
;
path parrB ; parrB :=
    parrA -- (0,-1) -- (2,-1) -- (2,-2) -- (4,0)
;
path p ; p := fullcircle scaled 2cm ;

fill (dashing (p, parrA, 25) && cycle)                withtransparency (1,.5) ;
draw (dashing (p, parrA, 25) && cycle)                withtransparency (1,.5) ;
fill (dashing (p, parrB, 25) && cycle) shifted (3cm,0) withtransparency (1,.5) ;
draw (dashing (p, parrB, 25) && cycle) shifted (3cm,0) withtransparency (1,.5) ;
\stopMPcode
```

In figure 18.8 we see the result. Of course how well if comes out depends on the definition but what is special here is that we use the double ampersand operator. That one will connect the paths without complaining about the end and being point not colliding. I suppose there was a good reason for making that a condition in the case of fonts, after all MetaFont is what it came from, but there is no real reason for it. It is a cheap extension anyway. At the same time I decided to add a native 'direction' operator. The number of extra bytes

in the binary is probably less than what is needed in memory to store the macro and the advantage is that we save an extra run over the path to reach the point we're consulting.[26]

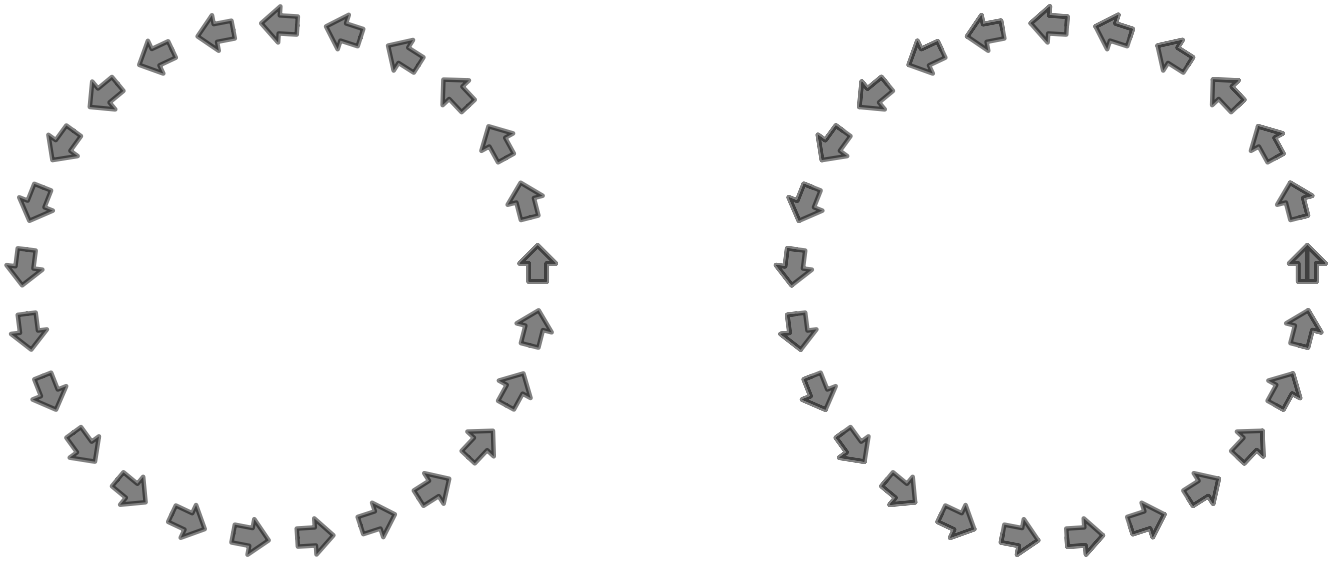


Figure 18.8 A somewhat related rendering.

In case you wonder why we need this feature, here is an argument:

```
\startMPcode
    path s ; s := fullcircle scaled 4cm ; pickup pencircle scaled 5mm ;

    draw (s shifted (0cm,0) && s shifted (3cm,0) && s shifted (6cm,0))
        withcolor "darkred" withtransparency (1,.5) ;

    currentpicture := currentpicture shifted (-8cm,0) ;

    draw s shifted (0cm,0)
        withcolor "darkblue" withtransparency (1,.5) ;
    draw s shifted (3cm,0)
        withcolor "darkblue" withtransparency (1,.5) ;
    draw s shifted (6cm,0)
        withcolor "darkblue" withtransparency (1,.5) ;

    currentpicture := currentpicture shifted (-8cm,0) ;

    nodraw s shifted (0cm,0) ;
    nodraw s shifted (3cm,0) ;
    nodraw s shifted (6cm,0) ;
    dodraw origin withcolor "darkgreen" withtransparency (1,.5) ;

\stopMPcode
```

The results are shown in figure ??. Which if the alternatives you prefer also depends on how you generate the shape. The `nodraw` variant for instance can be mixed with calculations without the need to revert to `hide`.

[26] In case you wonder, this is how the macro definition looks like: `vardef direction expr t of p = postcontrol t of p - precontrol t of p enddef ;`. Because points are searched from from the start there are two lookups needed. Normally this is no problem but Mikael and I are playing with really large paths, like those that come from drawing functions.

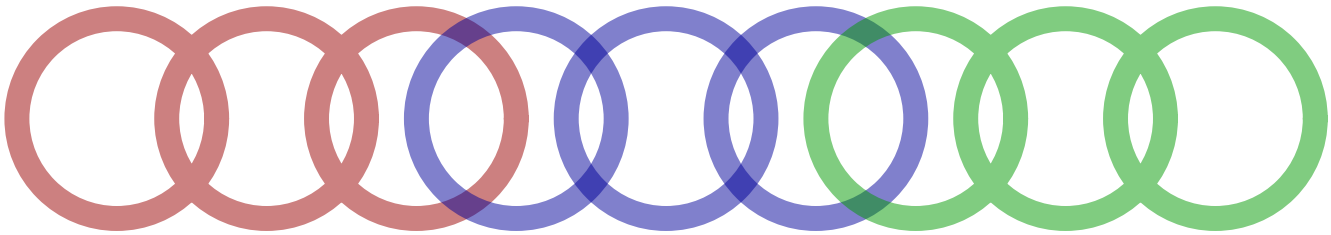


Figure 18.9 Do you see the difference?

Figure 18.10 demonstrates how far we've come. Mikaels fancy arrows nicely follow the shape of the function. Of course you need to make sure that these arrows are reasonably scaled. The definition of `dashing` demonstrates a few primitives that permits efficient iteration over a path and `arcpointlist` is sort of a path.

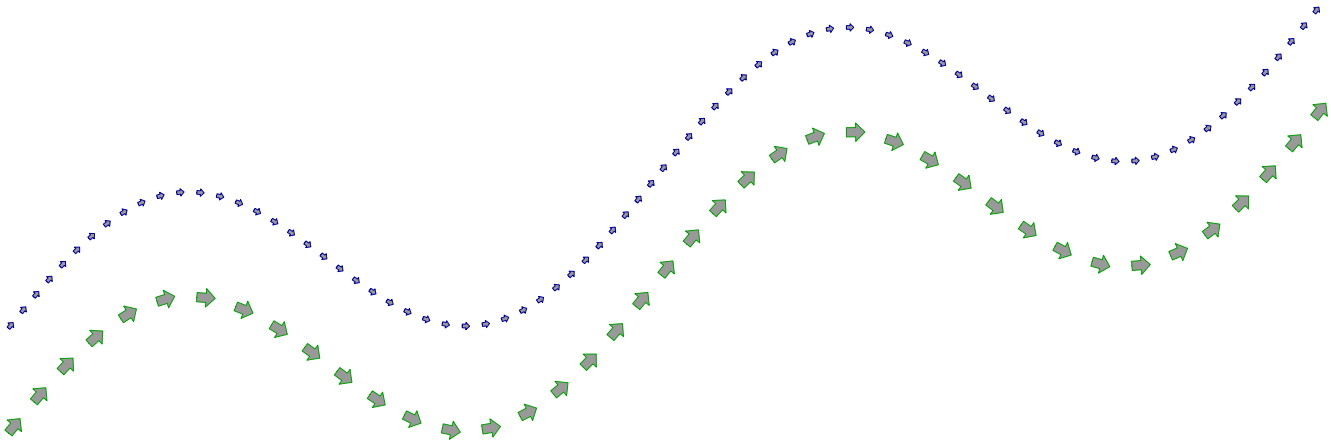


Figure 18.10 Advanced pseudo dashing.

```
\startMPcode
vardef dashing (expr pth, shp, stp) =
    for i within arcpointlist stp of pth :
        shp
            rotated angle(pathdirection)
            shifted pathpoint
        &&
    endfor nocycle
enddef ;

path e, p ; numeric n ;
e := (0,0) -- (0,-1) -- (2,-1) -- (2,-2) -- (4,0) -- (2,2) -- (2,1) -- (0,1) --
                                                                        (0,0) ;
n := 10 * bbwidth(e) ;
p := function(1,"x","x/4 + sin(x)",epsed(0.1),epsed(4*pi),0.01) scaled 2cm ;

fill (dashing (p, e scaled 2.5, n) && cycle) withcolor .6white    ;
draw (dashing (p, e scaled 2.5, n) && cycle) withcolor  darkgreen ;

currentpicture := currentpicture shifted (0,-2cm) ;

n := 20 * bbwidth(e) ;
fill (dashing (p, e, n) && cycle) withcolor .6white   ;
draw (dashing (p, e, n) && cycle) withcolor  darkblue ;
\stopMPcode
```

19 Dealing with math fonts

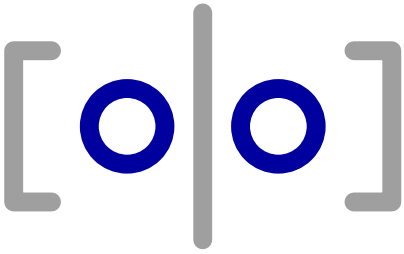
Introduction

Here we will explain some of the tricks that we apply to math fonts so that they not only work better with the LuaMetaTEX math engine but also look better, at least in our opinion. We will not show specific fonts because after all, who can complain about something that comes for free, but you can see whatever we do to make it work in action in ConTEXt where we setup these fonts. This is also a summary of what Mikael Sundqvist and I have been doing for a while now: improve the rendering of math, a rather enjoyable experience, also because we ran into humorous effects with and properties of fonts. Because we consider ourselves free from any conventions we could happily explore solution.

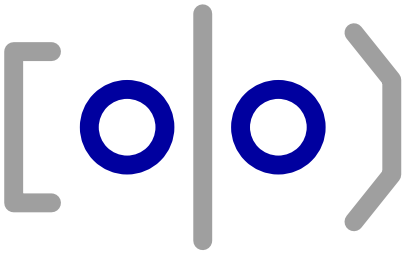
Fences

Fences come in two variants: fixed sizes and so called extensibles that are constructed from recipes that combine snippets that can partially overlap. Fenced material has an optional symbol at the start, an optional one at the end and zero or more symbols in the middle. Here we have all three:

Ideally these symbols scale in the same way, depending on the other context. This means that TEX first has to measure what sits in between but we will not dive into that. The left and right symbols are normally pairs like parentheses or braces, but any mix is possible. Ideally a font is designed with this in mind but unfortunately we see this:



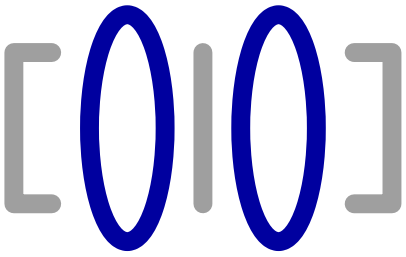
And even this:



It might be a side effect of the limited amount fo available slots in traditional math fonts that also resulted in non consistent sets in OpenType follow-ups and when one fonts does that more follow that approach.

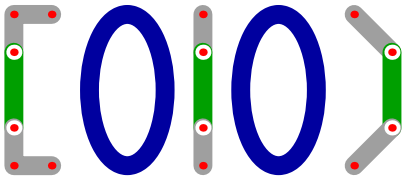
You can find rendering like this:

and this

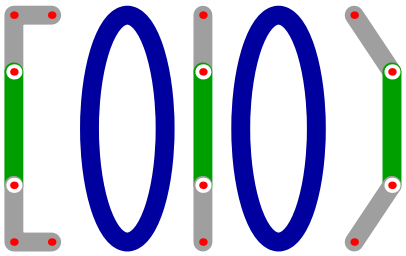


because a programmable language like TEX can use some tricks to force sizes: we just create some local fence which dimension is determined by some invisible rule. In LuaMetaTEX we can actually enforce dimensions and in ConTEXt we can filter specific sizes.

So how does this sizing work? A fence character starts out with the normal size but then a larger one is needed, the math engine will check if there is a larger variant. An OpenType font can provide these and in the engine that works out as following a linked list to a next size. When we run out of sizes there can be an extension recipe present where a fence is made from snippets pasted together. Normally that goes unnoticed because there is a little overlap between these snippets.



A larger fence will simply add more middle pieces, and it will not do as below:



Because we're talking of a deliberate design you cannot simply scale snippets and expect them to work out well visually. However, in a pure vertical case one actually could and in practice all these extensibles have (of course) vertical bars. Anyway, in the above example the larger middle piece actually is just several middle pieces overlapping.

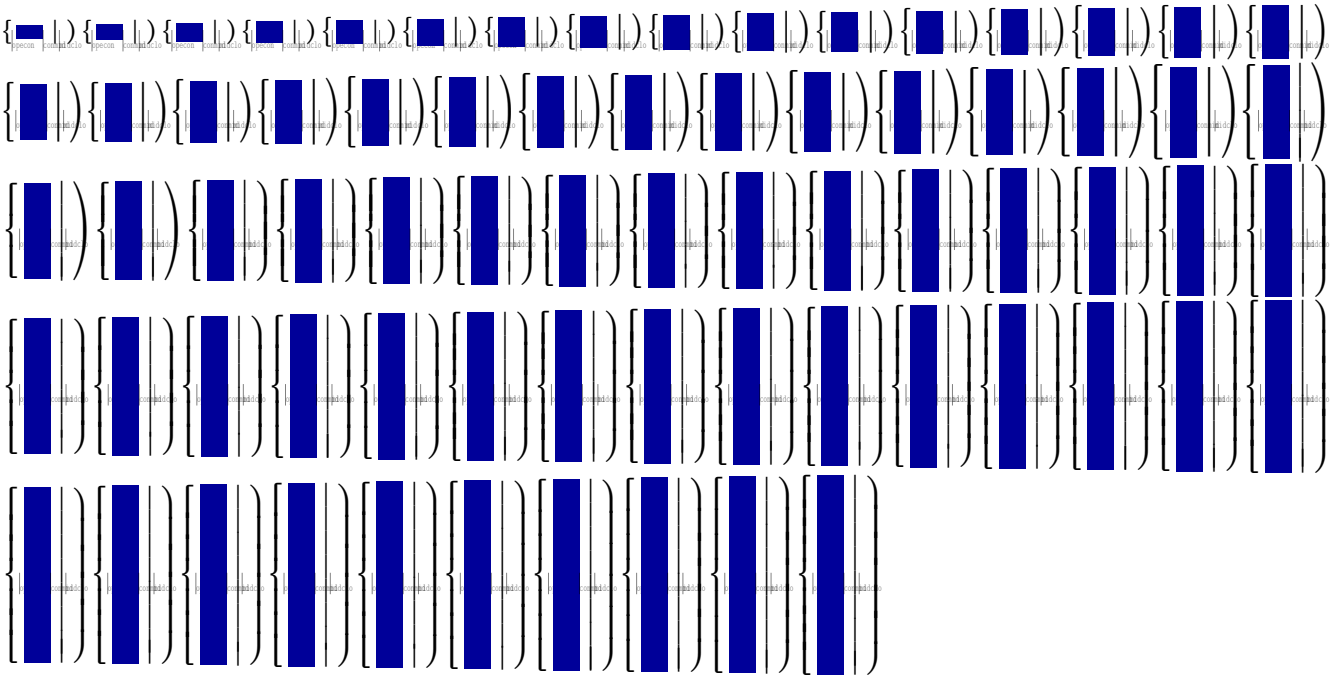
However, as we mentioned, fonts are not always consistent. First of all, when we run over the (increasing in size) variants we have discrete steps and you're lucky if a font has more than half a dozen. As soon as we end up with the extensible the size can be matched well.

So how do we compensate for misbehavior? There are two parameters in TEX then determine the matching: `\delimiterfactor` and `\delimitershortfall`. Plain TEX set them to 901 and 5pt which works okay in

most cases. In ConTEXt we set them to 1000 and 0pt and instead use the parameters `\UmathDelimiterPercent` and `\UmathDelimiterShortfall` that are bound fo fonts. In addition to that we use the `nooverflow` keyword with `\Umiddle` which makes sure that we always stay within the size of the outer fences. That just looks better.

In addition we can tweak the dimensions of glyphs and apply effects such as expanding so that we get a bit more consistent visual appearance. We can also signal that we should ignore sizes larger than a given index.

The next sequence show what happens in practice when we tell `\Umiddle` to never exceed the requested size. Because we start with stepwise sizes the first part of this sequence has no matching sizes. At some point we end up at the extensibles.



Its is worth noticing that we tried several alternative approaches. For instance what happens when we only use extensibles? In that case there will be no fit for the smaller ones because the at least two parts of an extensible can seldom completely overlap that much.

Actually, when we tested that we noticed that even in valid situations there can be strange overlap. At that time for instance the Lucida fonts had overlapping artifact in some curly braces which we found out when we tried to nil some of the larger, odd looking, step variants. Latin modern and some of the gyre fonts had unexpected jumps to larger sizes which made us decide to make the delimiter parameters font specific so that we could more easily adapt them: they basically became part of the math parameters of a font.

There are also inconsistencies in the perceived widths of glyphs used: often the bars are too thin. That can be solved by applying effects like scaling horizontally or vertically and cheating a bit with the dimensions. Another solution is that we ignore the variants after a certain size and force extensibles sooner but that of course needs to be tested for unwanted overlaps too. All these tricks combined make it possible to use math fonts with imperfect fences more or less reliable.

- Provide an equal amount of fixed size larger variants for all fences: assume arbitrary pairing.
- Because fonts have plenty room, provide some ten variants before going extensible.
- Try to make the variants and the extensibles similar in look.
- Ensure that the width of the vertical bar matches the design.
- Make sure that the odd entries in a extensible recipe don't overlap badly.

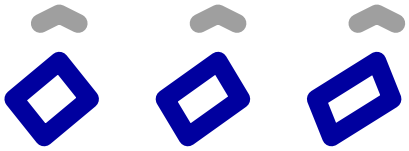
Accents

When traditional TEX showed up it was not that common to have pre-composed characters so when you needed something with an accept on top the way to go was to typeset the base character and position the accent on top using either the `\accent` primitive or some macro. It always was a compromise but eventually fonts with more assembled characters showed up. In OpenType fonts that operate in the Unicode domain we have even more characters but even there characters can be composed. However, anchors that help achieving this are part of the format. For text we use the mark features and for math we use the top anchor. Given that, why do we need to tweak it?

Here we have a base character with an accent on top. The character is upright and the accent gets positioned in the middle.

This doesn't work out well if we have a slanted or italic shape:

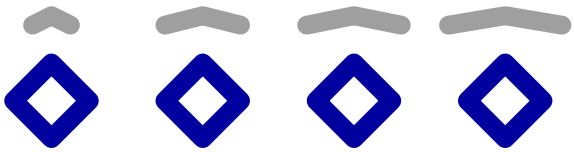
So we need to compensate, for instance like this:



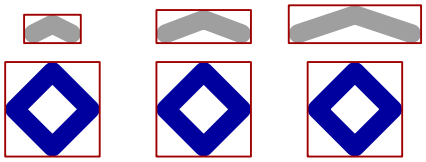
However, what does determine the right anchor point? From this example you can conclude that it is the top of the character. It is probably for that reason why the semi automated construction of Latin Modern and the Gyre fonts have quite some anchors that are rather bad: getting the anchors right is more a visual job than something that can be automated. The topmost point is not really the best one to focus on.

Here the topmost position is very off center. In for instance Latin Modern that means that on digits like 7 and 4 you get very weird anchoring. And this is why we have a tweak that just wipes all the anchors from an alphabet: most alphabets don't need them anyway and the engine will use the center when no anchor is defined. Just for the record: in traditional TEX engines the horizontal position is determined by the kern between a so called skew character and the base character. The font format has no anchor field but it has kerns, so this trick makes much sense.

We discussed vertical extensible that grow but horizontally we have accents that can grow. There are also a few horizontal fences like braces that have extensibles but we will cover that later. Accents are such that they only have a fixed set of variants and one problem is that there are often not enough of them. This means that the engine has to choose one that is reasonable.



In the example above the first two are acceptable but the third and fourth are not. Just imagine that there is a superscript or subscript involved. Here we apply another cheat: we lie about the dimensions. A glyph can have left and right margins that get subtracted when the accent analyzer tries to make a fit which means that we can sort of enforce the second solution. The ConTEXt font goodie files set the margins for some problematic characters because (of course) these are not part of OpenType math fonts. This is an optical issue mostly because the engine will not easily put a too wide one on top.



Watch how the larger accents also have a larger bounding box. That is all right but does interfere with a consistent makeup. The solution is simple: we use the ConTEXt dimension tweak to reposition accents and cheat with their height and depth to make sure that we get consistent rendering. We need to tweak anyway because sometimes accents have bad dimensions. The smallest one is actually a text accent and therefore can have properties that are inconsistent with its wider variants. This is typically a side effect of the fact that math accents and text accents are not considered to be different.

- Only add anchors to some (forward leaning!) italic shapes.
- Make extensibles as much a possible consistent with respect to dimensions.

Kerning

In a text font there are two mechanisms that influence the spacing between individual characters: kerning and italic corrections. In OpenType text fonts we have a more generalized relative positioning mechanism which can be seen as kerning. The italic correction well known to TEX users can be implemented as a positional font feature but very seldom is.

An OpenType math font has both kerning and italic correction. The kerning at the left top, left bottom, right top and right bottom of a glyph can be specified as a staircase and is used to position scripts. The italic correction is bit more curious and is applied is some cases. Keep in mind that in math a sequence of alphabetic characters does not make a word but represents a multiplication of ordinary symbols and thereby specific inter-atom spacing rules apply.

A traditional TEX font has kerns and italic correction and an OpenType font has staircase kerns and italic correction. For practical (space and time) reasons the widths of italic shapes in traditional math fonts are such that when you add the correction they kind of match the bounding box.

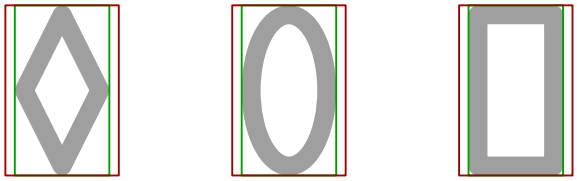
So, one way to suit both kind of fonts is to add the italic correction (often absent in OpenType glyphs but very present in traditional ones) as well as the staircase kerns (present in OpenType fonts and unknown to traditional fonts).

There are however some complications: what if a glyph has both? Which one is to be preferred? Should we try our luck? Even worse: in the traditional the italic correction is *always* added to the box that wraps a

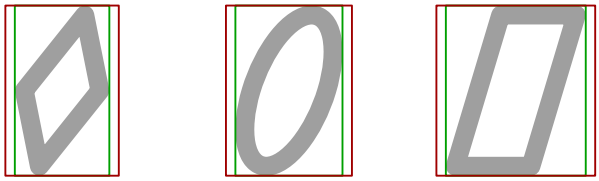
glyph but in some cases that correction gets removed.[27] But should we also remove a staircase kern? When we started with LuaMetaTEX it was a bit of a gamble because the specification only showed up later and improved over time.[28]

In LuaTEX we often have to code paths. That was done after other attempts to deal with this weak aspect of fonts worked for one font and not for the other. For instance at the time of this writing some fonts have italic corrections for upright characters! In LuaMetaTEX that model was changed into a detailed control model at the font as well as engine level. Later, when Mikael and I went over the fonts, usage of characters in math, atoms and spacing, we decided to kick out that detailed control and use more general control mechanisms assuming that we use OpenType fonts without bad italics. Whenever we had bad ones we could correct that in a so called font goodie file. In other words: no heuristics in the engine but fixed fonts. For that we always take Cambria as reference.

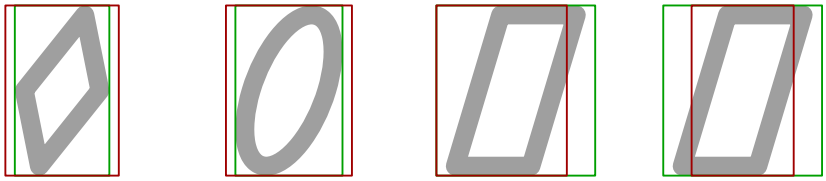
To be considered: should we finally turn italic correction into top and bottom kerns? Basically: model after Cambria? Then we can kick out code! We just assume font goodies.



Here we have three shapes and as usual they have some space at the left and right. A text font like Lucida is designed in such a way that no kerns between glyphs are needed but most text fonts have kerns. These compensate for these average acceptable side bearings.



In these slanted versions we get wider shapes but not always. For some shapes the amount of perceived spacing at the left top and right bottom increases and this is where we start thinking in terms of italic correction:



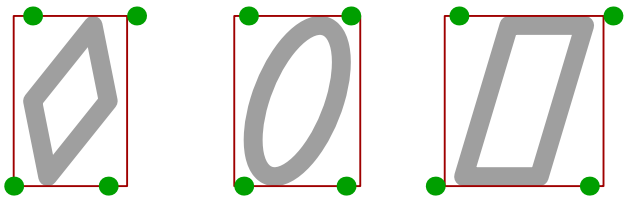
That way, when we put characters next to each other on the average words look better. But how about math: a superscript should be outside that bounding box and a subscript inside, assuming that we have a shape like this. It is unfortunate that the widely used f is the perfect candidate. When using that one as test case things can look great but kick in a g and it gets worse. The v and w are also a fertile playground. It is hard to come up with logic that satisfied all which is why glyph specific engine control was implemented (and later dropped). In the previous graphic the fourth shape also cheats on the left and yes, there are some fonts that do just that, which the of course interferes with prescripts.

[27] In LuaMetaTEX we never remove but compensate so that we can track what happens.

[28] This is true for much of OpenType which has the danger that bugs and side effects become features. Keep that in mind when you criticize solutions that early adopters came up with!

Now think of using shapes in formulas: some are put in sequence, in which case inter atom spacing is added so there is less danger of touching due to too a narrow width and TEXies somehow accepted that adding thin spaces every now and then is fine[29] But there's more than sequences: shapes end up in scripts, as degrees in radicals, above and below fraction rules, as fences and accents. Just assume the worst possible scenarios!

In these examples the italic correction at the right is the difference between the red and green box. There is no left italic correction and that is why OpenType math (driven by Cambria) has these four sets of kerns.



Here we show some possibilities for better anchoring but it will be clear that it is a compromise. Staircase kerns as in OpenType therefore have a set of kerns going up or down for not only the base character but also the one that ends up in a script as that one itself can have a 'problematic' shape.

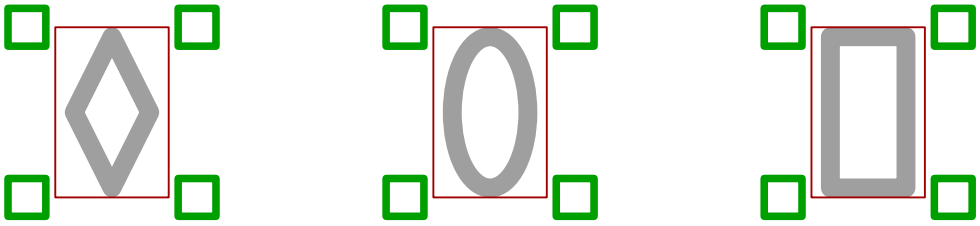
Our solution to this problem is to tweak dimensions and italic correction of problematic characters. We also can add four kerns that roughly compensate for tricky corrections needed. We therefore have more glyph properties than the official OpenType specification provides (cheaper and easier than staircase kerns that work on pairs of characters) . But we have more font parameters anyway, and with TEX being one of the main math renderers and TEXies always being to tune and tweak we think this is okay. It is better to have more control than to rely on (hard to fight) heuristics.

We're stuck with the fonts we have and (in the case of TEX fonts) the chosen traditional approach of dimensions and italic corrections (no staircase kerns) but the least we can ask is:

- Be consistent in dimensions and italic corrections. Get rid of limitations imposed by the 8 bit font era: we have plenty slots available and some more glyph properties as well. And if you compromise: make that clear.
- Don't just take the old properties but assume that OpenType math fonts are used in a modern OpenType TEX engine.
- Assume that any character is used in any combination: that is what made it hard to satisfy all needs at the engine end. Better play safe.

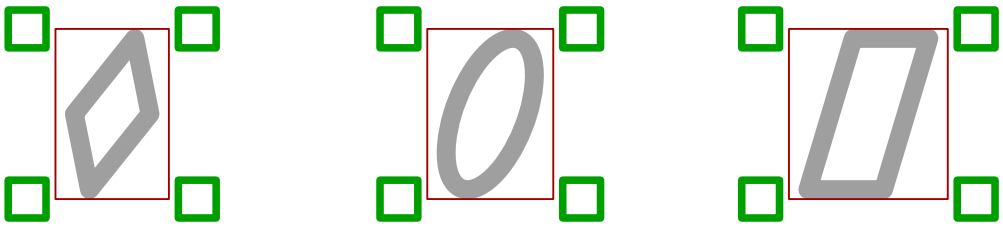
Scripts and primes

We start with showing a few shapes (in TEX speak: nuclei) with a different perceived spacing and with some items attached to the corners: scripts.

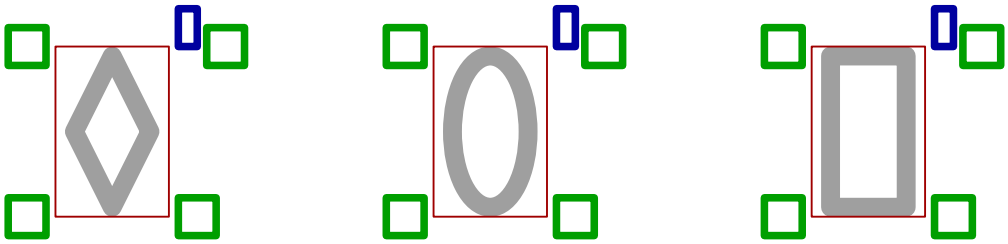


The green ones represent the super- and sub-, pre- and postscripts. According to what we discussed previously the anchoring depends on the shape

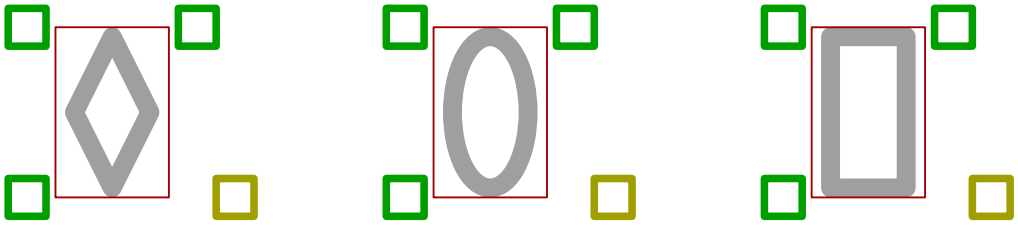
[29] We don't think so which is why we came up with a more granular inter atom spacing mode.



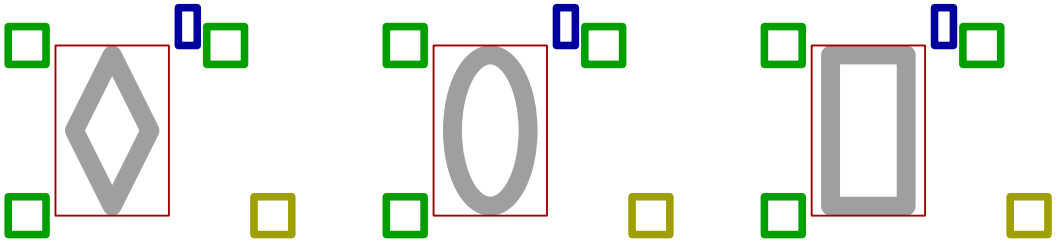
But we will not take that into account here because this time we focus on support for primes and indices. A prime normally is a narrow character that is positioned after the nuclues and sits in a similar position as a superscript. When it is present a subscript doesn't move. Here the blue rectangle represents the prime.



An index, here yellow, is a subscript that applies to the whole so that one does move. We cannot have a subscript and an index at the same time. The prescripts are not affected by primes and indices other than that the engine has to do more work in getting it right.



Of course we can have primes and indices at the same time. A prime actually belongs to the nuclues so that combination determines quite a bit of the vertical and horizontal spacing.



The problem with primes is that they were never part of the concept and OpenType math inherited that. Thanks to Unicode (and the math community not being convincing enough) we ended up with a prime being equivalent to minutes and double primes being minutes. The fact that we have triple and quadruple primes as well as reverse primes is a consequence of not having symbols for sub second indicators. And even the seconds are overloaded by meaning: time related and geographical.

What has this to do with primes? Well, it means that a font has a prime character that is already positioned at a certain distance from the baseline. But in most math fonts the script and scriptscript sizes aren't. It looks as if the assumption is that primes are treated like superscripts. But which one then gets superscripted at the text level? The text one or the script one? Unfortunately fonts are somewhat inconsistent in the sizing and positioning of primes and the math community not being convincing enough. It was a tough decision to make: in some fonts using the text prime have nice output and in other fonts the script one.

So, because in the traditional approach primes had to be manually positioned, be able deal with super and subscripts the old school approach was to make them active characters and pick up what follows in order to deal with this situation. It probably is the main reason why we have a somewhat special active character mechanism in math mode. In ConTEXt MkIV we followed a different approach, also because we wanted to deal with collapsing multiple primes to their rightful Unicode slot.

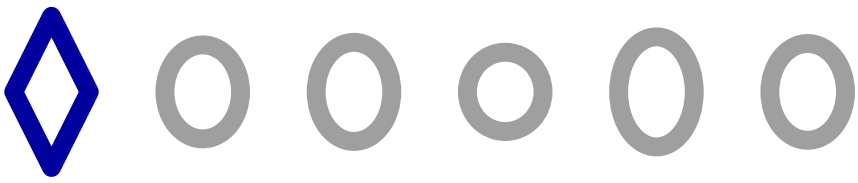
In LMTX and LuaMetaTEX we (need to) go a step further because we want proper inter-atom space as well as more fine tuned positioning. Primes became elements bound to a nucleus! We also wanted to solve this issue once and for all (in MkIV we has several methods but in LMTX we only have one left). We not only added a native prime element to nuclei but also introduced font parameters similar to those of superscripts. Just keep in mind that in LuaMetaTEX we don't need to worry about having a few more fields in a math node. There are not that many nodes involved and the amount of extra memory used can be neglected. Of course there is plenty of code added to deal with it so the binary definitely is larger due to it and the code way more complex. In fact we already introduced some missing parameters for the spacing related to prescripts. Furthermore we can tune the prime spacing relative to the scripts.

In the goodie files you can add fixes that relate to primes and most of them involved quite a bit of experimental. We can safely say that we spent quite some prime time on this issue.

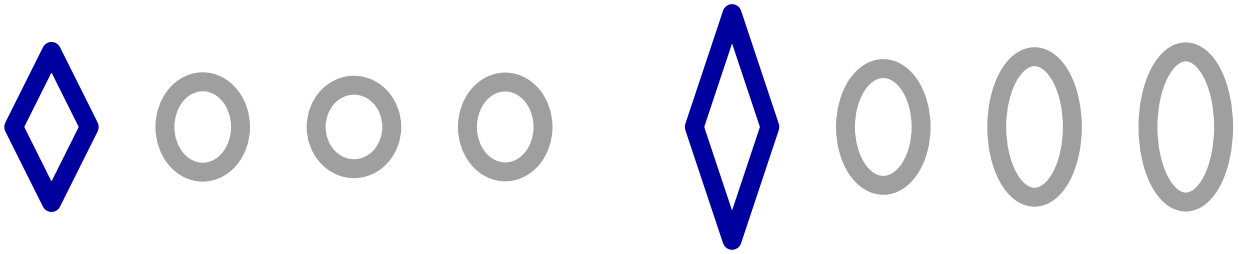
- It would be nice if fonts at least has prime shapes that are consistent because currently script one can look quite different (tilt and shape).
- Because primes are also minutes and seconds we probably have to accept the current situation and deal with it in the goodie files forever.

Two choices

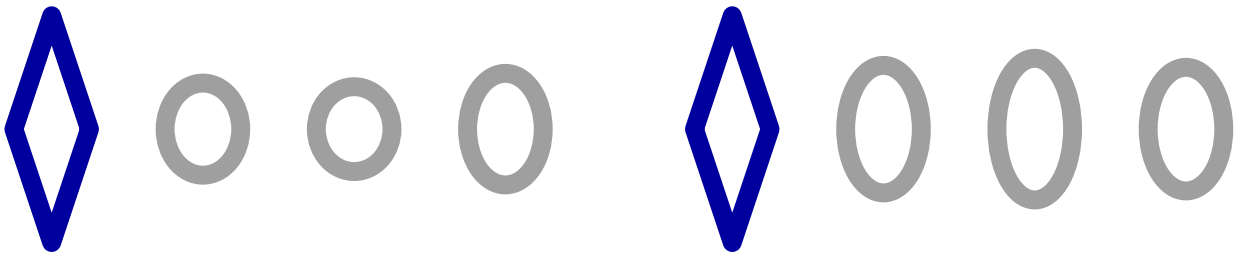
Imagine a situation like this: a sequence of characters with a leading left fence but nothing at the right. The fence is not really a fence but something that can grow and have its own super and subscripts, either after it or on top and below. We leave these scripts (aka limits) out of the discussion.



There is a visual aspect here, which can be illustrated from a variant:



One can argue that the larger characters in the second sequence rightfully trigger a larger blue variant. But for consistency one might actually go for:



Now imagine that you only have two choices? When do you go for the larger one? This situation occurs with so called large operators like integrals and summations. The original TEX fonts have two sizes of these characters. The smaller one is used in normal (text) math and the larger one in display math.

But what if, as in OpenType fonts there are more? There is a font parameter `DisplayOperatorMinHeight` that tells what size to use (as a minimum) in display mode. Because the TEX engines never had to make a decision the value of that variable in the Latin Modern and Gyre fonts is somewhat arbitrary. It is one of the variables that we need to adapt in the goodie files: we roughly bump from 1300 to 1800 to get what we are accustomed to in display mode.

So what are these large operators anyway? In traditional TEX they are just that: larger variants of smaller ones. In OpenType math however, they are extensibles. That means that when there are more variants and an extensible recipe it can grow on demand. That conflicts with the expectations of two sizes.

In order to support these two conflicting demands LuaMetaTEX provides so called left operators that either act upon their own, in which case we talk of 'auto' mode, or they can be told to have a specific size, or they can adapt, in which case they have the usual bogus right companion that ends the subformula. And, because they are implemented as fences but are not really fences there are some provisions for the scripts: they bind to the left fence and not the right one.

As side note: when we were experimenting with the usage of these (new) mechanisms we again ran into the race condition that is imposed by the algorithm that determines the size of fences: the delimiters target height is either a fraction of the total height of the sublist or the total height diminished by some constant amount. In LuaMetaTEX we therefore added the already discussed `\UmathDelimiterPercent` and `\UmathDelimiterShortfall` but these only kick in when we have a wrapped integral or such.

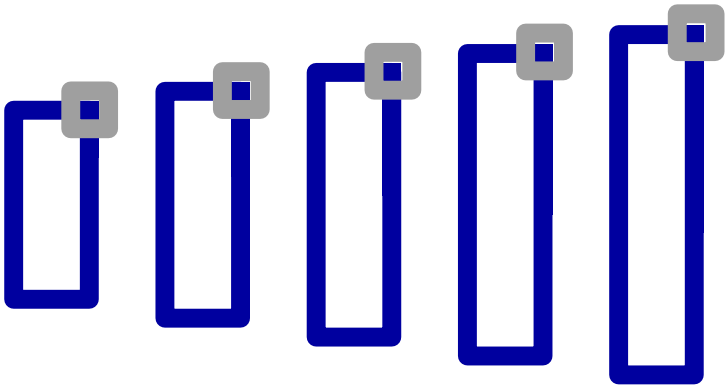
- We can't be too picky but it would be nice if fonts have `DisplayOperatorMinHeight` set to a more reasonable value. We have to tweak most font for this now (and probably forever).

To the point

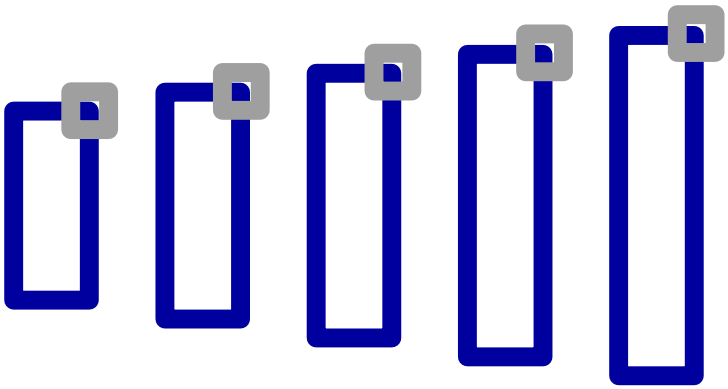
A nice feature of math fonts is that they have so called extensible characters. Fences can grow vertically and accents horizontally. When we run out of discrete steps in size, we end up with a recipe that build large characters from snippets as we discussed before.

The limitations in the size of fonts and the fact that the engine also had to impose some limits in the amount of used fonts and complexity of their usage made for some exceptions to the rule and, no pun intended, it might be why TEX has a concept of rules. Lines over and under something, arrows, fractions and radicals all use rules. It is interesting to notice that when OpenType fonts showed up the converted TEX fonts didn't make arrows and bars use the extension mechanism, thereby forcing the old school construction of them. This doesn't hurt ConTEXt much because we can implement proper characters using the virtual character mechanism and we did things different anyway (for instance we can kick in MetaFun replacements).

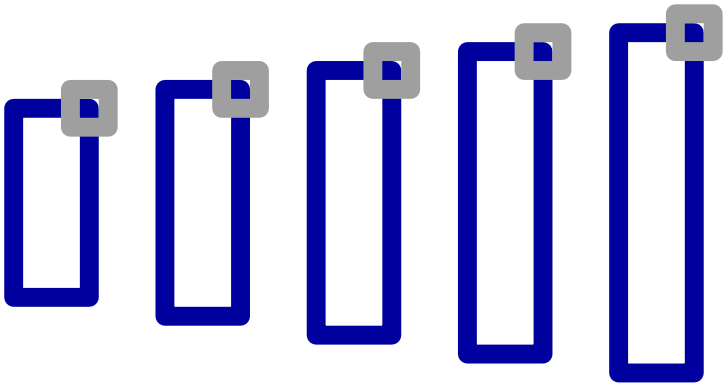
Nevertheless there is this curious in-between: radical. It comes in a set of fixed sized often sloped characters and then switches to an upright one. In all cases there is that horizontal rule attached that covers the nucleus. There is no mix of horizontal and vertical extensibles so there is no fancy end of rule thingie possible (the little hook that we learned to draw at school): it's just a stupid rule.



Instead of a rule, these examples show a simple rectangle attached to another one. The idea is that the center of that small one snaps onto the upper right corner of the large one. In these examples we show how it looks when that is not exact. At small sizes it goes unnoticed but at larger ones it becomes visible. Actually, when you start scaling up math you often see artifacts in shapes, which makes one wonder if high resolution screens are really used for proofing.



In the first example the effect is small because we make sure that the right top corner is square but in the second one we don't do that. Now it really becomes annoying, even if we remove the small errors in positioning.



We have seen different effects. The most common was an inaccurate font dimension that determines what the height of the rule is. That we can fix in the goodie file and we do that. Other artifacts were (in the stepwise sizes) characters that were so sloped that there was not possibility for properly attaching the rule (we noticed

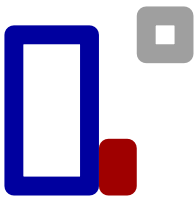
this best when checking the Lucida and Latin Modern fonts): one can cheat and move a little to the left but the thickness of the sloped line was not enough for that, so it could qualify as a design error but a design cannot simply be changed in such case: one has to compromise and assume no scaling (keep in mind that we scale on screen while on print one has to take a magnifying glass and reading math that way is not much fun).

Some of these extensible radicals have rather thin vertical lines but we can assume that their height is never such that this becomes too annoying. One complication in designing these shapes is that when one uses a font creator program it is unlikely to provide a math composer for testing. The same is true for overlapping the end glyphs of e.g. curly braces: the middle piece can get into the way when we have the smallest extensible and have that one kick in too soon due to the lack of stepwise sizes. It's hard to check that without an engine applying them to the extreme cases.

- Benefit from your high resolution screen and zoom in on the results. Take no risk.
- Play safe and create an top right corner that has some overlap and already starts out horizontally.
- If the design complicates rule attachments, move to an extensible sooner because that one can have a horizontal attachment point as part of the design.

Removing slack

In the process of optimizing the spacing we ran into several cases where a bit of spacing was added that eventually made that the resulting formula had small quite visible whitespace at the edges. Take for instance the little bit of kerning between a nuclues and a superscript.



Normally one can predict this situation but when components are glued together that are typeset independently the (in this case red) spacing remains. The engine is capable to remove that kind of slack at the left and right of the formula so that one gets a proper tight bounding box. There is a bit more logic on board now.

The fact that all kind of spacing gets added is one of the reasons for the more granular inter atom spacing: that way we could remove for instance the spacing added to the left and right of fractions (which is buried in the constructed box).

- There is not much we can advice here because it doesn't relate to fonts. However, given that Lucida text fonts have no kerning could make one wonder to what extend math fonts needs all these compensations.
- Some of the inter atom/script kerning is probably there as safeguard for italic shapes where staircase kerns are lacking. The question then is how the related font parameters themselves relate to staircase kerns. We suppose that this puts some stress on the font designer.

Initializing fonts

introduction

When a math font is initialized there are (as with any font) features applies. The only official one is the ssty feature that defines the substitution sets for script and scripscript size. In traditional TEX one has to set up multiple fonts because one can only have 256 characters and because one might want to map alphabets on the regular ascii slots. In an OpenType font all these alphabets are on the same font so basically one can do with three instances per size: text, script and scriptscript. What actually is done depends on the philosophy of the macro package but we will not discuss that here. So let's assume three instances in the default case. In LuaMetaTEX we can scale on the fly and therefore we can stick to one instance: as long as we know what the slots of the (optionally provided) smaller sizes are. Supporting this had quite some consequences for the engine as scaling happens all over the place (also think of font parameters) and we're talking two independent dimensions here: horizontal and vertical sizing.

In addition to this size related font feature ConTEXt provide some additional ones (and always had). Some are set up as regular features and some are driven by the font goodie features which organizes a lot of features under one umbrella. Font goodies always have been part of MkIV and LMTX.

also mention the non tweak related ones

wipeitalics

This tweak wipes the italics from one or more characters, most noticeable upright characters. Because locating the individual characters took too much time, we added the option complete alphabets.

`example`

wipeanchors

This tweak removes the top anchors from one or more characters, most noticeable upright characters. As with wiping italics, we often wipe complete alphabets.

`example`

accentdimensions

Configure the position of overbar, underbar, overbrace, underbrace, overparent, underparent, overbracket and underbracket. More? Added 220307.

addrules

I think we did not get overbars for some fonts (kpfonts and erewhon come to mind).

addscripts

Maybe something done for Ton? (`math-act`)

checkspacing

I don't know how it works. Seems there is not really any settings. But you once wrote

```
$\liminf$ \quad
$\limsup$ \quad [\sixperemspace] \quad
$x\sixperemspace x$

\setbox0\hbox{$x\sixperemspace x$} \showbox0
```

in an email (220118).

dimensions

We use this to change anchor points of accents, to raise/lower them, and of course to change bounding boxes and italic correction for many letters/symbols. We use it in bonum to resize the whole lower case fraktur alphabet (and maybe more should be added)

fixprimes

Maybe explained elsewhere? But it could belong here.

fixradicals

I don't know what it does. Could be: In some fonts the radicals are just positioned wrongly, and this moves them up/down to have something uniform to work with. (`math-act`)

kerns

A bit like staircase kerning. The 'topleft' and 'bottomright' come in handy, in particular the last one when it comes to the position of subscripts.

margins

We fake the width of some characters. Useful, for example, not to get too large/too small accents when using `\widehat` (also to get a uniform size over some alphabets or glyphs that often go togeter).

variants

Some fonts (lucida, xits, stixtwo) have calligraphic (chancery) and script (roundhand) alphabets. Of course not the same as default, and of course not the same ss0X.

wipecues

Only added to cambria and dejavu. Added 220327. I think it has to do with characters like 2061, 2062 and 2063. From the mails I think it has why does 2061 render a shape in some fonts so we need an 'wipe char' tweak (basically making it a zero dimensions symbol)

```
\mathspacingmode2
```

```
\showmakeup[mathglue]
$ \sin                 \Uchar"2061  (y) $ \quad
$ x                    \Uchar"2062   y  $ \quad
$ x                    \Uchar"2063   y  $ \par
$ \sin      \mathghost{\Uchar"2061} (y) $ \quad
$ x         \mathghost{\Uchar"2062}  y  $ \quad
$ x         \mathghost{\Uchar"2063}  y  $ \par
```

bigslots

Fences are chosen automatically to match what they surround. However, in traditional engines a fenced sub formula won't break across lines. In ConTEXt we have sveral st

1, 3, 5, 7 that could fit here, if we find a font where the linking is not present.

parameters

I don't know if all of them are parameters that typically moved from document to font level.

20 The three math sizes

What follow is a short explanation about how a TEX math engine deals with sizes. When typesetting a formula there are three different sizes assumed, as shown in the next example:

$$x^{x^x}$$

If we take each x and scale it to the same height we get the following. It is clear that the smallest size runs wider and looks a bit bolder which makes it more readable at the smaller size.

$$x\,x\,x$$

So why is this? The x is a math italic character and in traditional TEX that comes from what is called a math italic font. However, when TEX processes the list of intermediate math nodes, that x is specified as two numbers: a family and an index. A family is a collection of three references to fonts: they can be the same or different but whatever one chooses, they have to be scaled to what is expected. The initial TEX setup for instance makes a family from `cmmi10`, `cmmi7` and `cmmi5`. Each font has a slightly different design and when we typeset at 10 points no scaling takes place.

That all changed a bit when OpenType math fonts came around. In fact, there was only one such font to start with so that basically set the standard: Cambria from Microsoft. Instead of making three variants, the fonts contains upto three alternatives (although there is no real limit) for a specific shape. These variants are `ssty=1` and `ssty=2` where `ssty` refers to the alternative substitution font feature.

Fonts like Latin Modern also follow this model. So, in the case of an x, we have three options to choose from: the normal shape, or one of the `ssty` replacements. Because that font is mostly compatible with Computer Modern the x is like `cmmi10` and the smaller sizes like `cmmi` and `cmmi5` respectively. It is important to keep in mind that we are talking of one font! That means that in principle we only consume one family. Where with eight bit fonts we need quite some families, for instance for all these symbols, with OpenType fonts in a Unicode universe we can stick to one. Of course you can still use the more traditional model of multiple families, but just a copy of this one font will do then. In practice most symbols scale perfectly well to a smaller size so no special design size variants are provided and that makes an `ssty` mapping table relatively small: only for characters that benefit from it get an alternate.

It should also be noted that the alternates are not scaled down in their design, so actually the script x is in the font as if it were from `cmmi7` scaled up to 10 points. The font specifies two scaling factors but a macro package can decide to scale differently.[30] All this means that in TEX a family has three times the same font, the original (text), and two scaled down copies with `ssty=1` replacements (script and scriptscript). Although in practice an `ssty` feature will provide two alternates, there is no rule that mandates that. This introduces a dilemma:

> What are we supposed to do when there is a `ssty=1` but no `ssty=2`: do we use a scaled down text for scriptscript or do we scale down the script variant? Because all this is very much design related one can debate this.

[30] Many font parameters in an OpenType math fonts can be considered to have recommended values, and sometimes it makes sense to change them.

Actually this is a general question. In for instance the Latin Modern Math font there is an `aalt` feature where some glyphs have two and others three variants. Should choosing variant three when there are two available use the first or the last. The fact that in `ssty` there is some order, from small to even smaller, other alternate features provide just that: an alternative shape. It is good to keep in mind that OpenType provides mechanisms, just that, and these can be uses any way a designer wants. This is why we see the ligature mechanism being used for non-ligatures and ligatures being constructed by contextual replacements (that still can be single characters).

After dealing with math fonts for decades one gets a feeling of what is needed by users. And after implementing math font mechanism for eight bit fonts as well as OpenType fonts, one also gets an idea how much the ideal situation differs from reality. You can make a subsystem with plenty degrees of freedom but in the end when little gets used it only adds overhead. And math actually comes with some. In a traditional engine one has to deal with a limited number of families, which results in switching mechanisms (for instance to full range bold, not only alphabets) that also redefine math characters and reassign families. Because even a simple `$x$` already can trigger quite some initialization one has to be careful in coding this. For instance what happens with `{\bf $x$}` or `{\small $x$}` is not just a simple family switch. In for instance section titles switching style and size happens at the same time. I won't go into details how we dealt with all that in ConTEXt, lets stick to: we just have to make sure that the overhead is acceptable.

When working on LuaMetaTEX it was decided to introduce a more dynamic scaling model. First for regular text, later for math. There are several reasons for this. For text if can reduce the overhead of loading fonts at different sizes. It saves memory as well as load time, which involves scaling and passing to TEX, something that is noticeable for large fonts used for non Latin scripts.

This is also true for math but there we can go further. If instead of defining a font three times (per family) we pass the engine the information where a script and from that a scriptscript variant is located in the font, then we can delay scaling till such a glyph is needed. Every character has a `smaller` field and because we also support right to left math typesetting there's even a `mirror` field. It is up to the engine to deal with that. In principle we could now even turn family id's into font id's, but then we also need to handle the way families are used in math character specifications. For that reason mid 2023 this hasn't yet happened. We also don't want to abandon this nice TEX family concept, if only because we always need to support both models.

So, to summarize this:

- The traditional approach to sizes is to have three fonts with possibly different designs for shapes. A family consists of three different fonts.
- The OpenType approach is to have optional `ssty` alternates for different shapes at smaller sizes. A family can use the same font scaled differently.
- The LuaMetaTEX approach is to have one font for al three with size variants as part of the glyph definition. The engine replaces and scales on demand.

It will be clear that in the case of LuaMetaTEX, when scaling is delayed, the engine has to do more work, because not only glyphs scale but also related font parameters. And because we can also apply dynamic scaling of text to math, that means multiple times scaling every time a dimension is needed. Not only that, we also scale in two dimensions so for every math parameter we need to take that into account. It meant a rather fundamental overhaul of the code, on top of the plenty extra features that the LuaMetaTEX math engine provides. Of course we could not divert too much from the principe approach to math typesetting, which is why this all happened stepwise and why it is an option. The fact that we have dynamic scaling also means that we can have smaller than scriptscript sizes without much effort, but that is besides the point we want to make here. Anyway, when a math font is defined one can set a `compactmath` field as well as a `textscale`, `scriptscale` and `scriptscriptscale` when defining the font (via the Lua interface)

So far what we do is not that much different from what fonts have been doing since the beginning of TEX. Even the potential lack of ssty=2 dilemma is not unique: one can have design sizes eight bit fonts with holes too and it is actually harder to deal with that. However, there are other potential issues. Just like ssty is an alternate feature there can be more. And alternates are a great way out of endless discussions about preferred shapes. Some math fonts therefore have many alternates in the ssXX and cvXX feature namespace. Of course there is no agreement about this so there is our second dilemma:

> How do we deal with additional alternates, given that the distribution of alternative shapes is arbitrary as is their grouping.

Although one way to control this is using Unicode modifiers, that is not something that TEXies like to input. It still demands some agreement about modifier related to original shapes and it doesn't cover the plenty unspecified ones. And how portable is this?

If we assume that a math user wants consistency (which is quite an assumption) one solution is to specify additional substitutions when we load the font. But then we have a third dilemma:

> Which substitution comes first: the chosen style alternatives or the size related alternates?

One can argue that the order that features are defines in fonts also determine this accumulated substitutions. That makes a lot of sense but when you look at some specification of how scripts are supposed to be dealt with in OpenType you can find somewhat vague remarks about the order in which specific features are to be applied. However, in practice one can best follow the order in the font. Say that we want a different lowercase delta. There are two options:

1. The stylistic variant comes before the sizing one.
2. The sizing variant comes before the stylistic one.

Now, if fonts are well designed, when there is a different design for each size, there should also be one for the variant. Or the reverse, when there are different variants, there should be different sizes. In practice that means that there can be six different deltas. The replacement tables can be made such that they both take care of mapping. Think of these glyphs names being related:

```
delta        delta.ss03
delta.ssty1  delta.ss03.ssty1
delta.ssty2  delta.ss03.ssty2
```

So what if the two deltas are missing we first replace the sizes? We basically get this:

```
delta        delta.ss03
delta.ssty1  delta.ss03 (scaled)
delta.ssty2  delta.ss03 (scaled)
```

or this:

```
delta        delta.ss03
delta.ssty1  delta (scaled)
delta.ssty2  delta (scaled)
```

It depends bit om the machinery and the order of features in the font. It makes no sense to point fingers to fonts here, simply because what would be shown here could be different after a font gets updated.

In ConTEXt we have two choices of dealing with this. We can immediately apply features, or we can delay them which permits a mixture of different deltas. This is actually needed for calligraphic versus script alphabets as one document can have both while Unicode math combines them, which then means that we need to revert to a feature applied.[31]

Even if we delay there is an issue: in the three fonts per family approach we already have the ssty applied. So then we replace the delta based on the size replacement. So, we can end up with:

```
text         : delta              -> delta.ss03
script       : delta.ssty1 scaled -> delta.ssty1 scaled
scriptscript : delta.ssty2 scaled -> delta.ssty1 scaled
```

However, when we delay scaling and use one font, as described for LuaMetaTEX, we haven't yet applied ssty, so instead of pre-scaled replacements:

```
text         : delta
script       : delta scaled -> delta.ssty1 scaled
scriptscript : delta scaled -> delta.ssty2 scaled
```

We can first replace and then scale.

```
text         : delta    delta.ss03
script       : delta -> delta.ss03 -> scaled
scriptscript : delta -> delta.ss03 -> scaled
```

In the non-replaced delta case we would have had:

```
text         : delta
script       : delta.ssty1 -> scaled
scriptscript : delta.ssty2 -> scaled
```

The difference is that because we haven't yet done the scaling and ssty replacement we check against the original Unicode and when there is a lacking delta.ss03.ssty1 at that size, we just scale down the text size delta.ss03. One can argue that we then might violate the lookup because maybe there was on purpose no alternate but how to distinguish design from error in fonts?

Of course when replacement is delayed one can also again consult the ssty table and sort of fix the font runtime but there is a catch there: you end up with a scaled glyph (scriptscript) in a differently scaled (text) font so you get different dimensions and although the ConTEXt backend can handle that it is confusing and not pretty. One has to use the virtual font mechanism and in fact the glyph we're dealing with can already be virtual (tweaked).

There are more cases where confusion can kick in. Examples are primes where can have different shapes and positioning in text compared to the script sizes, and accents, where text and replacements have either or not zero widths. To some extend it is no real surprise because one needs a math engine that can expose it when making the font. And after a while the font can't change any more because it's being and there are many TEX users who expect the same output after years. I can imagine that a text editor that basically assumes one specific math font being used (or expects other math fonts to be set up the same way) can provide a list of magic key combinations that choose some shape and/or provide some graphical user interface. I remember presentations at BachoTEX where was demonstrated how a desktop publishing application deals

[31] In ConTEXt we have a variant on this where we have a dedicated additional alphabets to this which also permits for taking calligraphic or script from other fonts, because not all fonts have both.

with alternates and every version got different or additional interfaces for that. In such a case a feature and interface depend on each other, something that is not the case in a TEX ecosystem: there we just access whatever we want. Of course there we also need some interface but there are at least some traditions in place, like naming characters or hiding complex and inconsistent choices in macros.

21 Constants

Strings don't really fit into the concept of TEX. There everything we input and store is tokens and nodes, so when you define a macro like

```
\def\foo{foo}
```

you don't store a string but a tokenlist with three tokens:

control sequence: foo					
770644	11	102	letter	f	U+00066
772818	11	111	letter	o	U+0006F
725549	11	111	letter	o	U+0006F

We have three single byte characters but end up with 32 bytes memory used because we have a linked list with a housekeeping initial token; such a token has a value (operator & operand) as well as a pointer to a next token. This is quite ok because whenever we need that macro the body has to be interpreted and it already being tokenized is what makes TEX fly.

There are occasions where the expansion of a list that itself can contain references to macros produces a new list in which case copies are being made. Take this:

```
\def\oof{foo}
\def\foo{foo \oof}
```

control sequence: foo					
770768	11	102	letter	f	U+00066
759797	11	111	letter	o	U+0006F
771222	11	111	letter	o	U+0006F
771289	10	32	spacer		
773152	144	0	call		oof

When `\foo` is expanded, the macro body is pushed onto the input stack and traversed and when `\oof` is seen, that one gets pushed and processed. No copy is needed. Now take this:

```
\def\oof{foo}
\edef\foo{foo \oof}
```

control sequence: foo					
770989	11	102	letter	f	U+00066
773269	11	111	letter	o	U+0006F
773331	11	111	letter	o	U+0006F
772809	10	32	spacer		
770765	11	102	letter	f	U+00066
770641	11	111	letter	o	U+0006F
770773	11	111	letter	o	U+0006F

Here `\foo` gets the expanded result but again `\oof` got pushed onto the stack. This doesn't involved copying either but there is still the pushing and popping input overhead. So when does copying occur? Here is an example:

```
\def\oof{oof}
\def\ofo{ofo}
\def\foo{\begincsname \oof:\ofo\endcsname}
```

control sequence: foo					
773051	140	2	cs name		begincsname
773286	144	0	call		oof
771249	12	58	other char	: U+0003A	
770528	144	0	call		ofo
772779	80	0	end cs name		endcsname

When a csname is checked, the engine needs to construct a string in order to access the hash table. Here is what happens:

- everything upto the `\endcsname` is collected
- in the process macros are expanded (with pushing and popping input) and the expanded tokens are appended to the result
- when we're okay that list get converted to a string
- that string is used as lookup into the hash

Normally we're okay but when there is some unexpected unexpandable token (an assignment, node generator, protected macro, etc.) the collection stops and the list so far is recycled. This process is quite efficient, as is everything TEX, but given that going from token list to string involved some utf8 juggling too there definitely is some overhead.

In ConTEXt we use csname checking and usage quite a lot. The first line is the traditional way. It has the disadvantage that it creates an hash entry with alias `\relax` if there is no such name. That is why e-TEX came up with the test as in the second line. In LuaTEX we introduced `\lastnamedcs` so that we don't have to construct the mentioned) token list again which saves time. The fourth line is similar to the first line but doesn't create a new command.

```
\csname      \namespace\key\endcsname                                  ...
\ifcsname    \namespace\key\endcsname \csname \namespace\key\endcsname ... \fi
\ifcsname    \namespace\key\endcsname \lastnamedcs                     ... \fi
\begincsname \namespace\key\endcsname
```

One of the things all versions of ConTEXt have in common (right from the start) is that we use this namespace model consistently. In MkIV we changed the subsystem that deals with this: it's more flexible and uses less memory but it also has way more overhead. But on the average performance is about the same so users didn't notice that.

There is however a trick to speed this up a bit. In the 360 page LuaMetaTEX manual we expand macros like `\namespace` and `\key` 4.3 million times (beginning of June 2023). Because Mikael Sundqvist and I are in the middle of some math magic, we also checked his 300 page math book, and that also does it 4.2 million times (the gain was about 0.5 seconds). The upcoming math manual has some 1.2 million. How come that we have so many expansions? First of all we use abstraction when possible and that means that there's plenty of checking of options and some constructs fall back on parent classes (sometimes more that two times up the parent chain). Also, we often have three macros to expand:

```
\ifcsname\namespace\currentinstance\key\endcsname
```

But these have an important property: their body is basically a string. Nothing in there needs expansion and if it does, it's an indication of rubish that doesn't contribute for a valid csname anyway. Once we know that we can improve performance:

```
\cdef\oof{oof}
\cdef\ofo{ofo}
\def\foo{\begincsname \oof:\ofo\endcsname}
```

So, `\cdef` (or `\constant\edef` flags the macro as being a constant that doesn't require expansion. For the record, when you define that macro having arguments it just becomes an `\edef`.

control sequence: foo

760115	140	2	cs name		begincsname
760055	147	0	constant call		oof
771555	12	58	other char	: U+0003A	
760200	147	0	constant call		ofo
772428	80	0	end cs name		endcsname

Here we define the two macros as constant ones which in practice means that they are just macros but also indicates that in some scenarios we can directly use their body. Now when in this csname construction we do this instead:

- everything upto the `\endcsname` is collected
- in the process macros are expanded (with pushing and popping input) but when we have a constant we add reference token when there is more than one body token, otherwise the expanded tokens are appended to the result
- when we're okay that list get converted to a string and in that stage we just convert the referenced body of the constant
- that string is used as lookup into the hash

So, instead of immediately injecting an expanded body of a macro that needs no expansion we inject a reference and use that later on for the conversion into characters. On the 4242938 times in LuaMetaTEX (at the time of writing this) this trick gives the following results.

```
\edef\foo{xxxx} \begincsname\foo\endcsname    0.37
\cdef\foo{xxxx} \begincsname\foo\endcsname    0.28
\edef\foo{xxxx} \ifcsname\foo\endcsname\fi    0.53
\cdef\foo{xxxx} \ifcsname\foo\endcsname\fi    0.35
```

And here for an existing command (`\relax`):

```
\edef\foo{relax} \begincsname\foo\endcsname   0.55
\cdef\foo{relax} \begincsname\foo\endcsname   0.36
\edef\foo{relax} \ifcsname\foo\endcsname\fi   0.62
\cdef\foo{relax} \ifcsname\foo\endcsname\fi   0.36
```

When I used that trick in for instance some font switching macros it also had some gain. For instance 200000 times `\it` went from 0.60 down to 0.54 seconds but it is unlikely that in a document one does that many font switches.[32]

[32] There are a few more places where constants can gain a little but those don't add up much.

In practice other operations play a role, so here we might also benefit from the data being in the cpu cache but on the manual I gained a decent .2 seconds. One can question if on a 8.5 second run this is worth the trouble. However, in this particular manual we spend 3.5 seconds on font processing, some 1.5 seconds on the backend and have a unique MetaPost graphics on every page. We spend more time in Lua than in TEX! On 4 seconds TEX, these .2 seconds is some 2.5 gain, and it might actually be even more percent wise.

In case one wonders why I spend time on this, one reason is that the last decade I was not that impressed by performance gains of a single core and TEX is a single core process. I also can't afford the latest greatest laptops and definitely don't want to contribute more e-waste. Also, with TEX and friends running on virtual machines and competing for resources (memory, cpu and disk or network drives) any gain is good gain. Of course it is also fun to improve LuaMetaTEX and this string-like property has always bothered me.[33]

[33] I did some experiment with a native string register but that made no sense because then tokenization in other places takes a toll. With the mentioned constants we don't pay that price.

22 Active characters

Each character in TEX has a so called category code. Most are of category 'letter' or 'other character' but some have a special meaning, like 'superscript' or 'subscript' or 'math shift'. Of course the backslash is special too and it has the 'escape' category.

A single character can also be a command in which case it has category 'active'. In ConTEXt the | is an example of that. It grabs an argument delimited by yet another such (active) bar and handles that argument as compound character.

From the perspective of ConTEXt we have a couple of challenges with respect to active characters.

- We want to limit the number of special symbols so we only really have to deal with the active bar and tilde. Both have a history starting with MkII.
- There are cases where we don't want them to be not active, most noticeably in math and verbatim. This means that we either have to make a sure that they are not active bit in nested exceptions, for instance when we flush a page halfway verbatim, made active again.
- In text we always hade catcode regimes to deal with this (which is actually why in LuaTEX efficient catcode tables were one of the first native features to implement. This involves some namespace management.
- In math we have to fall back on a different meaning which adds another (meaning) axis alongside catcode regimes: in math we use the same catcode regime as in text so we have a mode dependent meaning on top of the catcode regime specific one.
- In math we have this special active class/character definition value "8000 that makes characters active in math only. We use(d) that for permitting regular hat and underscore characters in text mode but let them act as superscript and subscript triggers in math mode.
- Active characters travel in a special way trough the system: they are actually stored as macro calls in token lists en macro bodies. This normally goes unnoticed (and is not that different from other catcodes being frozen in macros).

So far we could always comfortably implement whatever we wanted but sometimes the code was not that pretty. Because part of the LuaMetaTEX project is to make code cleaner, I started wondering if we could come up with a better mechanism for dealing with active characters especially in math. Among the other reasons were: less tracing clutter, a bit more natural approach, and less intercepts for special cases. Of course we have to be compatible. Some first experiments were promising but as usual it took a while to identify all the cases we have to deal with. At moments I wondered if I should go forward but as I stepwise adapted the ConTEXt code to the experiment there was no way back. I did however reject experiments that out active characters in the catcode table namespaces.

In LuaTEX (and its predecessors) internally active characters are stored as a reference to a control sequence, although a \show or trace will report the character as 'name'. For example:

```
\catcode `!=\activecatcode
\def !{whatever} % we also have \letcharcode
\def\foo{x!x}
```

is stored as (cs, cmd, chr):

control sequence: foo						
720480	11	120	letter	x	U+00078	
760160	144	0	call			whatever
770492	11	120	letter	x	U+00078	

However, when we want some more hybrid approach, a text versus math mix, we need to postpone resolving into a control sequence. Examples are macro bodies and token registers. When we flag a character (with amcode) as being of a different catcode than active in math mode, we get the following:

```
\amcode`! \othercatcode
\catcode `!=\activecatcode
\def !{whatever}
\def\foo{x!x}
```

control sequence: foo					
771525	11	120	letter	x	U+00078
770660	13	33	active char		
761004	11	120	letter	x	U+00078

The difference is that here we get the active character in the body of the macro. Interesting is that this is not something that parser is prepared for so the main loop has now to catch active characters. This is no big deal but also not something to neglect. The same is true for serialization of tokens.

Other situations when we need to be clever is for instance when we try to enter math mode. In math mode we want the (in text) active character as math character and a convenient test is checking the mode. However, when we see $ we are not yet in math mode and as TEX looks for a potential next $ we grab a active character it should not resolve in a reference to an command. The reason for that is that when TEX pushes back the token (because it doesn't see a $) we need it to be an active character and not a control sequence. If it were a control sequence we would see it as such in math mode which is not what we intended. It is one of these cases where TEX is not roundtrip. Similar cases occur when TEX looks ahead for (what makes a) number and doesn't see one which then results in a push back. Actually, there are many look ahead and push back moments in the source.

```
text: \def\foo{x|!|x}

\meaningasis\foo

\luatokentable\foo

$x\foo x$ \foo
```

text:

\def \foo {x|!|x}

control sequence: foo					
728820	11	120	letter	x	U+00078
771014	13	124	active char		
772567	12	33	other char	!	U+00021

760035	13	124	active char		
771276	11	120	letter	x	U+00078

xx|!|*xx* x!x

```
math: $\gdef\oof{x|!|x}$

\meaningasis\oof

\luatokentable\oof

$x\oof x$ \oof
```

math:

\global \def \oof {x|!|x}

control sequence: oof					
770811	11	120	letter	x	U+00078
770523	13	124	active char		
773084	12	33	other char	!	U+00021
771283	13	124	active char		
686462	11	120	letter	x	U+00078

xx|!|*xx* x!x

```
toks: \scratchtoks{x|!|x}

\detokenize\expandafter{\the\scratchtoks}

\luatokentable\scratchtoks

$x\the\scratchtoks x$ \the\scratchtoks
```

toks:

x|!|x

token register: scratchtoks
<no tokens>

xx|!|*xx* x!x

A good test case for ConTEXt is:

```
\def\foo{x|!|x||x}

 x|!|x||x + \foo
$x|!|x||x + \foo$
```

Here we expect bars in math mode but the compound mechanism applied in text mode:

x!x–x + x!x–x

$x|!_{\text{fcord}}|x||x_{|\text{arbinumvar}}+x|!_{\text{fcord}}|x||x$

So the bottom line is this:

- Active characters should behave as expected, which means that they get replaced by references to commands.
- When the \amcode is set, this signal the engine to delay that replacement and retain the active character.
- When the moment is there the engine either expands it as command (text mode) or injects the alternative meaning based on the catcode. There we support letters, other characters, super- and subscripts and alignment codes. The rest we simply ignore (for now).

Of course you can abuse this mechanism and also retain the character's active property in text mode by simply setting the \amcode. We'll see how that works out. Actually this mechanism was provided in the first place to get around the "8000 limitations! So here is another cheat:

```
\catcode `^ \othercatcode       % so a ^ is just that
\amcode  `^ \superscriptcatcode % but a ^ in math signals a superscript
```

So, the a in \amcode stands for both 'active' and 'alternative'. As mentioned, because we distinguish between math and text mode we no longer need to adapt the meaning of active commands: think of using \mathtext in a formula where we leave math mode and then need to use the text meaning of the bar, just as outside the formula.

In the end, because we only have a few active characters and no user ever demanded name spaces that mechanism was declared obsolete. There is no need to keep code around that is not really used any more.

Internally an active character is stored in the hash that also stores regular control sequences. The character becomes an utf string prefixed by the utf value of 0xFFFF which doesn't exist in Unicode. The \csactive primitive is a variant on \csstring that returns this hash. Its companion \expandactive (a variant on \expand) can be used to inject the related control sequence. If \csactive is not followed by an active character it expands to just the prefix, as does \Uchar"FFFF but a bit of abstraction makes sense.

23 Accuracy

One of the virtues of TEX is that it can produce the same output over a long period. The original engine only uses integers and although dimensions have fractions but these are just a way to present then to the user because internally they are scaled points.

`\dimexpr .4999pt : 2 \relax`	0.24994pt
`\dimexpr .4999pt / 2 \relax`	0.24995pt
`\scratchdimen .4999pt \divide\scratchdimen 2 \the\scratchdimen`	0.24994pt
`\scratchdimen .4999pt \edivide\scratchdimen 2 \the\scratchdimen`	0.24995pt
`\scratchdimen 4999pt \divide\scratchdimen 2 \the\scratchdimen`	2499.5pt
`\scratchdimen 4999pt \edivide\scratchdimen 2 \the\scratchdimen`	2499.5pt
`\scratchdimen 4999pt \rdivide\scratchdimen 2 \the\scratchdimen`	2500.0pt
`\numexpr 1001 : 2 \relax`	500
`\numexpr 1001 / 2 \relax`	501
`\scratchcounter 1001 \divide\scratchcounter 2 \the\scratchcounter`	500
`\scratchcounter 1001 \edivide\scratchcounter 2 \the\scratchcounter`	501

The above table shows what happens when we divide an odd integer or for that matter odd fraction. Note the incompatibility between `\numexpr` and `\dimexpr` on the one hand and `\divide` on the other. This is why in LuaMetaTEX we have the `:` variant that does the same integer divide (no rounding) as `\divide` does, and why we have `\edivide` that divides like an expression using the `/`. The `\rdivide` only makes sense for dimensions and rounds the result.

As soon as one start calculating or comparing accumulated values one can run into the values being a few scaled points off. This means that when one tests against a criterium it might be that some range comparison is better. The most likely place for that to happen is in the output routine and when special constructs like floats, tables and images come into play. Just like not every number can be represented in a float (double), we saw that dividing an odd integer can give some unexpected rounding as part of the integer is considered a fraction. So, in practice, even when the calculations are the same, there is a certain unpredictable outcome from the user perspective: "Why does it fit here and not there?" Well, we can be a few scaled points off due to some not entirely round-trip calculation.

When TEX showed up it came with fonts and in those times once a font was released it was unlikely to change. But today fonts do change. And changes means that a document can render differently after an update. Of course this is an argument for keeping a font in the TEX tree but even then updating is kind of normal. Take math: the fact that fonts often have issues makes that we need to tweak them and some tweaks only get added when we run into some issue. If that issue has been there for a while we are incompatible.

Hyphenation patterns are another source of breaking compatibility but normally they change little. And here one can also assume that the user want words to be hyphenated properly. Even with such fundamental changes as a syllable being able to move to the next line, it is often unlikely that the paragraphs gets less or more lines. I bet that users are more worried about the impact on vertical rendering that has consequences for page breaks that for lines coming out differently (hopefully better).

So, what are other potential areas in addition to slight differences due to division, fonts and patterns? We now enter the world of LuaMetaTEX and ConTEXt. As soon as one starts to use Lua code, doubles show up. It means that we can do calculation with little loss because a double can safely hold the maximum dimension (in scaled point). However, mixing 64 bit doubles at the Lua end with 32 bit integers in the engine can have

side effects. As soon as set some property at the TEX end using Lua rounding takes place. Of course we can do all calculations like TEX does, but that would have too much of an impact on performance.

So, going back and forth between TEX and Lua can introduce some inaccuracies creeping in but as long as it is consistent, there is no real issue. It mostly involves fonts and especially the dimensions of characters: the width, height and depth but when one uses the xheight as relative measure there is also some influence on for instance interline spacing, offsets and such.

So how can fonts make a difference? In ConTEXt there are two ways to use fonts: normal mode and compact mode. In normal mode every size is an instance, where the dimension properties of characters are scaled. In compact mode we use one size and delegate scaling to the engine which means that we end up with the (usual) 1000 being scale 1 kind of calculations. In the end a font with design size of 10bp (most fonts) scaled to 12pt normal is not behaving the same as a 10pt setup where a 12pt size is scaled on demand. First there is the scaling from 10bp loaded font to the 10pt used font that gets passed to TEX. Here we have to deal with history: defining a font in pt points is quite normal. Then applying a 1200 scale (later divided by 1000) in the engine again involves some rounding to integers because that is what is used internally. I will come back to this later.

The main conclusion to draw is that normal mode and compact mode come close but give different results. We can come closer when we a more accurate normal mode. In order to limit the number of font instances we normally limit the number of digits (also in compact mode but there accuracy comes a little cost). There is a pitfall here: While TEX can happily work with any resolution, the backend has to make sure that embedded fonts get scaled right and that (in the case of pdf) we compensate for drift in the page stream, because there character widths determine the advance and these are in (often rounded) bp (big postscript points). Especially when we enable font expansion drift prevention comes with a price as there we are dealing with real small difference in dimensions.

As an experiment I played with clipping measures in the engine which boils down to rounding the last digit but that didn't work out well. For simple text we can get normal and compact mode identical but kick in some math (many parameters involved), font expansion and/or protrusion, additional inter-character kerning and so on, and one never get the same output. Keep in mind that we are not talking visual differences here, although there can be cases. More think of due to a slightly different vertical spacing triggering a different page break, for instance when footnotes are involved. In ConTEXt the line height (and therefore derived parameters) is defined in terms of the xheight so even a few scaled points off makes a difference.

At the user level, currently compact mode is enabled with:

```
\enableexperiments[fonts.compact]
```

It works quite okay already for years (writing end 2023) in most scenarios but there might be cases where existing code still needs to be adapted, which is no big deal. The additional overhead is compensated by loading less font instances and a smaller output file. In some cases documents actually process faster and it definitely pays of for large fonts (cjk) and demanding mix size feature processing.

A more accurate normal mode is set by:

```
\enableexperiments[fonts.accurate]
```

but it doesn't bring much. It was introduced in order for Mikael Sundqvist and me to compare and check math tweaks, especially those that depend on precise combinations of glyphs. We temporary had some additional control in the engine but after experiments and comparing variants the decision was made to remove that feature.

We ran experiments with large documents where different versions were overlaid and depending on scenarios indeed there can be differences, but when there are chapters starting on new pages and when vertical spacing has stretch, there are not that many differences. When you compare the so called `tuc` files you might notice small difference is position tracking but these values are seldom used in a way that influences the rendering of text, line and page breaks.

To come back to the bp vs pt issue. Among the options considered are moving the character and font properties from integers to doubles, but that would impact the memory footprint quite a bit. Another idea is that compact mode goes 10bp instead of 10pt but that would not help. One bp is 657817 scaled points and one pt is 655360 sp. The ratio between them is 1.0037490844726562, so a TEX scale 1200 effectively becomes 1204.499, and assuming rounding to an integer we then get 1204. So in the end we get a less fortunate number instead of 1200 and it's not even accurate. Therefore this option was also rejected. For the record: an intermediate approach would have been to cheat: use an internal multiplier (the shown ratio) and although it is not hard to support, it also means that at the Lua end we always need to take this into account, so again a no-go.

In the end the only outcome of this bit of 'research' has been that we can have accurate normal font handling (which is not that useful) and have two additional divide related primitives that might be useful and add some consistency (and these might actually get used).

24 Characters in math

This chapter goes into some more details about math characters but after some remarks goes on about about discretionaries. Traditionally TEX users enter ascii characters mixed with commands and expect to get the right visual representation. Because there are plenty math characters outside the ascii range this means that most are accessed by a command. However, Unicode changes that: when an editor can show the character there is no reason not to use that feature. In that case we end up with utf characters in the input. In this perspective is it important to realize that there is a distinction between such a direct utf character and a command, especially when it is defined as follows:

```
\Umathchardef\mathcharacterf 0 \mathordinarycode `f

\startformula
    f = \mathcharacterf = \Uchar"1D453
\stopformula
```

This gives the expected:

$$f = f = f$$

The ascii `f` will eventually become character `U+1D453` but let's not worry here about how that is done; what is more important is that this character has some extra properties. Just like the definition of the command we use a primitive `\Umathcode` that registers that we have an ordinary and that the origin is family zero, we also need to make sure that the `U+1D453` has those properties. The way it works is that the engine injects a math noad with a math character nucleus and when it does that it needs to resolve the family and the class. Depending on the style the family will resolve in a text, script or scriptscript font. The class determines the spacing and some specific engine behavior.

In ConTEXt and therefore in LuaMetaTEX we go as step further. There we also have dictionary fields, which makes it possible to adapt properties like the class as we like after the user has entered them. This (experimental) features relates to the fact that often Unicode math, TEX character names, and usage doesn't really reveal what the character is about and if it is needs to have class binary, relation or something else. If the command does carry some meaning it gets lost when we end up with these injected math characters. In LuaMetaTEX we do carry more around. Because this is experimental and evolving we stick to mentioning that there is for instance a primitive `\Umathdictdef` that does what `\Umathchardef` does but expects three additional numbers: properties, group and index.

In LuaMetaTEX we try to be as detailed as possible when we resolve and store references to characters in the math nodes (noads), even if the engine itself doesn't always need that information, for instance: for handling a single character superscript we don't need to know its class.

This detour was needed in order to understand the following: discretionaries in math mode. In LuaMetaTEX we are already more tolerant with respect to what can end up in a discretionary and math discretionaries have been supported for a while now. In the next examples, class 2 is used: binary.

```
test $ \dorecurse{50}{a \discretionary class 2 {$+$}{$+$}{$+$} } b$ test
```

test $a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a +$
$+ a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + b$ test

But this is not nice: we need to enter math mode in the three snippets and likely also need to make sure that we do that in the right style. So, that was why we can now also do this:

```
test $ \dorecurse{50}{a \mathdiscretionary class 2 {<}{>}{=} } b$ test
```

test $a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a <$
$> a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = a = b$ test

We can wrap this in a command:

```
\def\weirdrelation{\mathdiscretionary class 2 {<}{>}{=}}
```

but this is not what we want when we are talking `+` and `-` which are candidates for repetition. And these are entered as utf character so there is indication of them being treated special. This is why LuaMetaTEX has a new vector `\hmcode` where one can trigger specific characters to become discretionaries.

```
\hmcode"002B=1 % +
\hmcode"2212=1 % -
```

```
test $ \dorecurse{50}{a + b - } c$ test
```

test $a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b -$
$- a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a +$
$+ b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b -$
$- a + b - a + b - a + b - a + b - a + b - a + b - a + b - c$ test

Setting bit one of the code will enable this feature. But as usual with TEX and math there is a pitfall. Take this (unusual) example:

```
\hmcode"1D453=1 % we trigger promotion to discretionary
```

```
test $\dorecurse{50}{a \Umathchar 2 0 "1D453 b} b$ test
```

test $a\,f\,ba\,f$
$f\,ba\,f$
$f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,bb$ test

We see the `f` being repeated but also notice that the italic correction disappears because that is what happens in the line break. But in math this correction is actually part of the width (we've written plenty about that over the years). However, when we set bit two of the code, the correction is moved into the discretionary:

```
\hmcode"1D453=3 % we carry the italic correction along
```

```
test $\dorecurse{50}{a \Umathchar 2 0 "1D453 b} b$ test
```

test $a\,f\,ba\,f$
$f\,ba\,f$
$f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,ba\,f\,bb$ test

So, where characters need to retain their family and class, we also need to make sure that we retain the fact that a character is to be automatically repeated at a line break. The reason why this ended up in the engine while it could be delegated to a callback is that we do need to process discretionaries in math anyway and also want to avoid it when we're not at the outer level. And because we already carry around all kind of options with noads and glyphs it was not that hard to support this.

It is a bit of a side track but discretionaries in LuaMetaTEX are a bit more permissive anyway. Take this:

```
\dorecurse{20}{%
    xxxxxx
    \discretionary {>>} {<<} {==}
    xxxxxxxxxx
}
```

xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx ==
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx ==
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx

Here we depend on the tolerance and stretch settings in order to not overflow the text boundaries. But how about the next:

```
\dorecurse{20}{%
    xxxxxx
    \discretionary
      {>\hskip0pt plus 5pt>}
      {<\hskip0pt plus 5pt<}
      {=\hskip0pt plus 5pt=}
    xxxxxxxxxx
}
```

xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = =
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = = xxxxxxxxxx xxxxxx = =
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx

This time we have some glue in the snippets. But we can even do the next trickery, where we can stretch the boxed content after the line break routine has done it work. It is this mechanism that we use deep down in the math engine too.

```
\dorecurse{20}{%
    xxxxxx
    \discretionary
        {\uleaders \hbox to 2em{>\hss>}\hskip0pt plus 10pt minus 5pt}
        {\uleaders \hbox to 2em{<\hss<}\hskip0pt plus 10pt minus 5pt}
     %  {\uleaders \hbox to 2em{=\hss=}\hskip0pt plus 10pt minus 5pt}
        {==}
  % xxxxxxx
    xxxxxxxxxx
}
```

xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx ==
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx ==
xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx
xxxxxx == xxxxxxxxxx xxxxxx == xxxxxxxxxx

So, in some way, extending the math engine lets features trickle back into the text engine and vise versa. It is all about seeing (weird) opportunities because it is often after playing with this that one sees more potential.

25 Standardizing math fonts

25.1 Introduction

ConTEXt has always had a good support for the typesetting of mathematics. ConTEXt MkII uses the pdfTEX engine and hence traditional (Type1) fonts. Several math fonts are available, specifically designed to work seamlessly with TEX. ConTEXt MkIV, the successor version, utilizes the LuaTEX engine, providing support not only for traditional fonts but also for OpenType Unicode math fonts. Unlike the XETEX engine, which interpreted these new fonts in a manner similar to traditional TEX fonts, LuaTEX adheres more closely to the (unfortunately somewhat vague) OpenType specification.[34] When new fonts appeared, some were more like the traditional fonts, others more like OpenType Unicode math fonts. This leads to difficulties in achieving consistent results across different fonts and might be one reason that the Unicode engines are not yet used as much as they probably should.

In autumn 2021 we started to discuss how to improve the typesetting of OpenType Unicode mathematics, and it was natural to go on and do this for the LuaMetaTEX engine, and hence for ConTEXt LMTX. Since then, we have been engaging in daily discussions covering finer details such as glyphs, kerning, accent placement, inter-atom spacing (what we refer to as math microtypography), as well as broader aspects like formula alignment and formula line breaking (math macrotypography). This article will primarily focus on the finer details. Specifically, we will explore the various choices we have made throughout the process. The OpenType Unicode math specification is incomplete; some aspects are missing, while others remain ambiguous. This issue is exacerbated by the varying behaviors of fonts.

We make runtime changes to fonts, and add a few additional font parameters that we missed. As a result, we deviate from the standard set by Microsoft (or rather, we choose to interpret it in our own way) and exercise the freedom to make runtime changes to font parameters. Regarding this aspect, we firmly believe that our results often align more closely with the original intentions of the font designers. Indeed, the existence of "oddities" in these fonts may be attributed to the lack of an engine, during their creation, that supported all the various features, making testing difficult, if not essentially impossible. Within ConTEXt LMTX, we have the necessary support, and we can activate various helpers that allow us to closely examine formulas. Without them our work would not have been possible.

Ultimately, we hope and believe that we have made straightforward yet effective choices, rendering the existing OpenType Unicode math fonts usable. We hope that this article might be inspiring and useful for others who aim to achieve well-designed, modern math typesetting.

25.2 Traditional vs. OPENTYPE math fonts

There is a fundamental difference between traditional TEX math fonts and OpenType Unicode fonts. In the traditional approach, a math setup consists of multiple independent fonts. There is no direct relationship between a math italic x and an ˆ on top of it. The engine handles the positioning almost independently of the shapes involved. There can be a shift to the right of $\hat{x}$ triggered by kerning with a so-called skew character but that is it.

A somewhat loose coupling between fonts is present when we go from a base character to a larger variant that itself can point to a larger one and eventually end up at an extensible recipe. But the base character and that sequence are normally from different fonts. The assumption is that they are designed as a combination. In an OpenType font, variants and extensibles more directly relate to a base character.

[34] See https://learn.microsoft.com/en-us/typography/opentype/spec/math

Then there is the italic correction which adds kerns between a character and what follows depending on the situation. It is not, in fact, a true italic correction, but more a hack where an untrue width is compensated for. A traditional TEX engine defaults to adding these corrections and selectively removes or compensates for them. In traditional TEX this fake width helps placing the subscript properly while the italic correction is added to the advance width when attaching subscripts and/or moving to the next atom.

In an OpenType font we see these phenomena translated into features. Instead of many math fonts we have one font. This means that one can have relations between glyphs, although in practice little of that happens. One example is that a specific character can have script and scriptscript sizes with a somewhat different design. Another is that there can be alternate shapes for the same character, and yet another is substitution of (for instance) dotted characters by dotless ones. However, from the perspective of features a math font is rather simple and undemanding.

Another property is that in an OpenType math font the real widths are used in combination with optional italic correction when a sequence of characters is considered text, with the exception of large operators where italic correction is used for positioning limits on top and below. Instead of abusing italic corrections this way, a system of staircase kerns in each corner of a shape is possible.

Then there are top (but not bottom) anchor positions that, like marks in text fonts, can be used to position accents on top of base characters or boxes. And while we talk of accents: they can come with so-called flat substitutions for situations where we want less height.

All this is driven by a bunch of font parameters that (supposedly) relate to the design of the font. Some of them concern rules that are being used in constructing, for instance, fractions and radicals but maybe also for making new glyphs like extensibles, which is essentially a traditional TEX thing.

So, when we now look back at the traditional approach we can say that there are differences in the way a font is set up: widths and italic corrections, staircase kerns, rules as elements for constructing glyphs, anchoring of accents, flattening of accents, replacement of dotted characters, selection of smaller sizes, and font parameters. These differences have been reflected in the way engines (seem to) deal with OpenType math: one can start with a traditional engine and map OpenType onto that; one can implement an OpenType engine and, if needed, map traditional fonts onto the way that works; and of course there can be some mix of these.

In practice, when we look at existing fonts, there is only one reference and that is Cambria. When mapped onto a traditional engine, much can be made to work, but not all. Then there are fonts that originate in the TEX community and these do not always work well with an OpenType engine. Other fonts are a mix and work more or less. The more one looks into details, the clearer it becomes that no font is perfect and that it is hard to make an engine work well with them. In LuaMetaTEX we can explicitly control many of the choices the math engine makes, and there are more such choices than with traditional TEX machinery. And although we can adapt fonts at runtime to suit the possibilities, it is not pretty.

This is why we gradually decided on a somewhat different approach: we use the advantage of having a single font, normalize fonts to what we can reliably support, and if needed, add to fonts and control the math engine, especially the various subsystems, with directives that tell it what we want to be done. Let us discuss a few things that we do when we load a math font.

25.3 Getting rid of italic corrections

OpenType math has italic corrections for using characters in text and large operators (for limits), staircase kerns for combining scripts, and top anchor for placement of accents. In LuaMetaTEX we have access to more features.

Let's remind ourselves. In a bit more detail, OpenType has:

- `n \typ{italic correction} is injected between characters in running text, but: a sequence of atoms is {\em not} text, they are individually spaced. \stopitem \startitem n italic correction` value in large operators that reflects where limits are attached in display mode; in effect, using the italic correction as an anchor.
- `Top anchors` are used to position accents over characters, but not so much over atoms that are composed from not only characters.
- `Flat accents` as substitution feature for situations where the height would become excessive.
- `Script and scriptscript` as substitution feature for a selection of characters that are sensitive for scaling down.

This somewhat limited view on math character positioning has been extended in LuaMetaTEX, and we remap the above onto what we consider a bit more reliable, especially because we can tweak these better. We have:

- `Corner kerns` that make it possible to adjust the horizontal location of sub- and superscripts and prescripts.
- Although `flat accents` are an existing feature, we extended them by providing additional scaling when they are not specified.
- In addition to script sizes we also have `mirror` as a feature so that we can provide right to left math typesetting. (This also relates to dropping in characters from other fonts, like Arabic.)
- In addition to the `top anchors` we also have `bottom anchors` in order to properly place bottom accents. These are often missing, so we need to construct them from available snippets.
- An additional `extensible italic correction` makes it possible to better anchor scripts to sloped large operators. This is combined with keeping track of `corner kerns` that can be specified per character.
- Characters can have `margins` which makes it possible to more precisely position accents that would normally overflow the base character and clash with scripts. These go in all four directions.
- In order to be able to place the degree in a radical more precisely (read: not run into the shape when there is more than just a single degree atom) we have `radical offsets`.

There are plenty more tuning options but some are too obscure to mention here. All high level constructors, like fences, radicals, accents, operators, fractions, etc. can be tuned via optional keyword and key/values at the macro end.

We eliminate the italic correction in math fonts, instead adding it to the width, and using a negative bottom right kern. If possible we also set a top and bottom accent anchor. This happens when we load the font. We also translate the italic correction on large operators into anchors. As a result, the engine can now completely ignore italic corrections in favor of proper widths, kerns and anchors. Let us look at a few examples.

The italic f is used a lot in mathematics and it is also one of the most problematic characters. In TEX Gyre Bonum Math the italic f has a narrow bounding box; the character sticks out on both the left and right. To the right, this is compensated by a large amount of italic correction. This means that when one adds sub- and superscripts, it works well. We add italic correction to the width, and introducing a negative corner kern at the bottom right corner, and thus the placement of sub- and superscripts is not altered. Look carefully at the bounding boxes below.

original tweaked

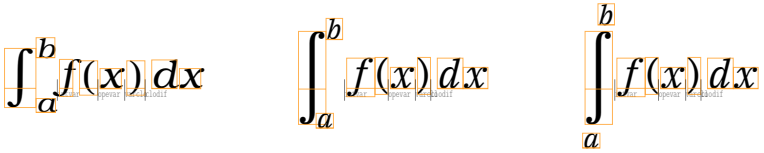
Compare with Lucida Bright Math, which comes with staircase kerns instead of italic correction. We convert these kerns into corner kerns.

original tweaked

For characters that stick out to the left, we also increase the width and shift the glyph to ensure that it does not stick out on the left side. This prevents glyphs from clashing into each other.

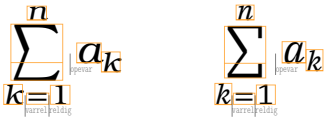
original tweaked

As mentioned, for the integral, one of the most common big operators, the limits are also placed with help of the italic correction. When the limits go below and on top, proper bottom and top anchor points are introduced, calculated from the italic correction. (The difference in size of the integral signs is a side effect of the font parameter `DisplayOperatorMinHeight` being tweaked, as we'll discuss more later. OpenType fonts can come with more than two sizes.)



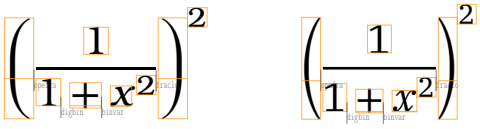
original tweaked, nolimits tweaked, limits

Compare these integrals with the summation, that usually does not have any italic correction bound to it. This means that the new anchor points end up in the middle of the summation symbol.



original tweaked

We also introduce some corner kerns in cases where there were neither italic corrections nor staircase kerns. This is mainly done for delimiters, like parentheses. We can have a different amount of kerning for the various sizes. Often the original glyph does not benefit from any kerning, while the variants and extensibles do.

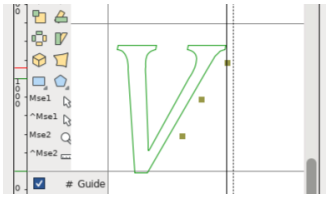


original tweaked

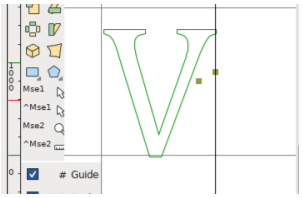
Note also the different sizes of the parentheses in the example above. Both examples are set with `\left(` and `\right)`, but the font parameters are chosen differently in the tweaked version. Font designers should have used the opportunity to have more granularity in sizes. Latin Modern Math has four, many others have steps in between, but there is a lack of consistency.

25.4 Converting staircase kerns

We simplify the staircase kerns, which are often somewhat sloppy and seldom complete (see figure below), into more reliable corner kerns. It's good enough and looks better on the whole. We also avoid bugs that way.



italic V



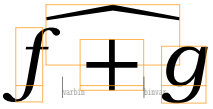
upright V

25.5 Tweaking accents

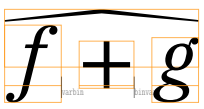
We ignore the zero dimensions of accents, simply assuming that one cannot know if the shape is centered or sticks out in a curious way, and therefore use proper widths with top and bottom anchors derived from the bounding box. We compensate for negative llx values being abused for positioning. We check for overflows in the engine. In case of multiple accents, we place the first one anchored over the character, and center the others on top of it.

We mentioned in an earlier TugBoat article that sometimes anchor points are just wrong. We have a tweak that resets them (to the middle) that we use for several fonts and alphabets.

Some accents, like the hat, can benefit from being scaled. The fonts typically provide the base size and a few variants.



original



tweaked

The only fonts we have seen that support flattened accents are Stix Two Math and Cambria Math.

stix two

cambria

If you look carefully, you notice that the hats over the capital letters are not as tall as the one over the lowercase letter. There is a font parameter `FlattenedAccentBaseHeight` that is supposed to specify when this

effect is supposed to kick in. Even though other fonts do not use this feature, the parameter is set, sometimes to strange values (if they were to have the property). For example, in Garamond Math, the value is 420.

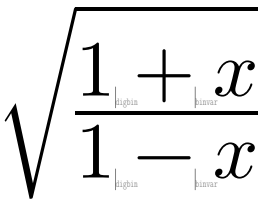
We introduced a tweak that can fake the flattened accents, and therefore we need to alter the value of the font parameter to more reasonable values. We communicated to Daniel Flipo, who maintains several math fonts, that the parameter was not correctly set in Erewhon math. In fact, it was set such that the flattened accents were used for some capital letters (C in the example below) but not for others (A below). He quickly fixed that. The green rules in the picture have the height of `FlattenedAccentBaseHeight`; it did not need to be decreased by much.

Erewhon, not fixed

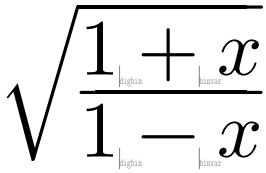
Erewhon, fixed

25.6 Getting rid of rules

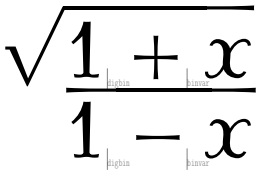
We get rid of rules as pseudo-glyphs in extensibles and bars. This also gives nicer visual integration because flat rules do not always fit in with the rest of the font. We also added support for this in the few (Polish) Type 1 math fonts that we still want to support, like Antykwa Toruńska.



With rule



With glyph



Antykwa

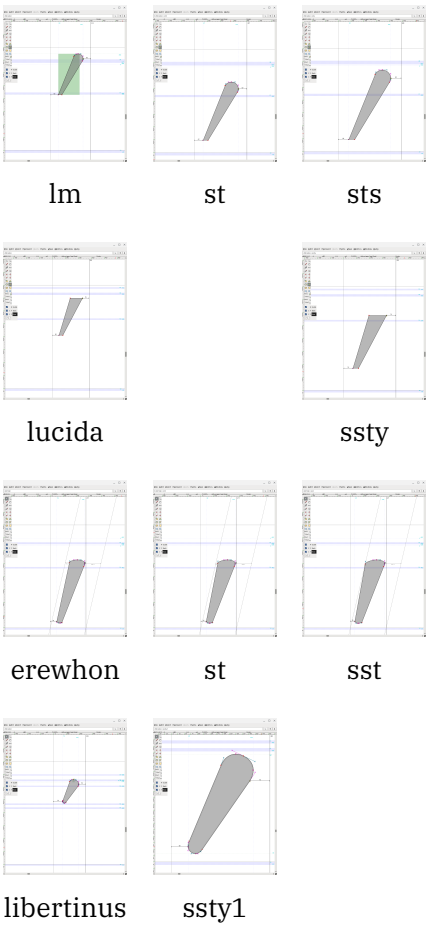
Here is an enlarged example of an Antykwa rule. Latin Modern has rounded corners, here we see a rather distinctive ending.

25.7 Tweaking primes

We make it no secret that we consider primes in math fonts a mess. For some reason no one could convince the Unicode people that a ‘prime’ is not a ‘minute’ (that is, U+2032 PRIME is also supposed to be used as the symbol for minutes); in case you’d like to argue that “they often look the same”, that is also true for the Latin and Greek capital ‘A’. This lost opportunity means that, as with traditional TEX fonts, we need to fight a bit with placement. The base character can or cannot be already anchored at some superscript-like position, so that makes it basically unusable. An alternative assumption might be that one should just use the script size variant as a superscript, but as we will see below, that assumes that they sit on the baseline so that we can move it up to the right spot. Add to that the fact that traditional TEX has no concept of a prime, and we need some kind of juggling with successive scripts. This is what macro packages end up doing.

But this is not what we want. In ConTEXt MkIV we already have special mechanisms for dealing with primes, which include mapping successive primes onto the multiple characters in Unicode, where we actually have

individual triple and quadruple primes and three reverse (real) primes as well. However, primes are now a native feature, like super- and subscripts, as well as prescripts and indices. (All examples here are uniformly scaled.)



Because primes are now a native feature, we also have new font parameters `PrimeShiftUp` and `PrimeShiftUpCramped`, similar to `SuperscriptShiftUp` and `SuperscriptShiftUpCramped`, which add a horizontal axis where the primes are placed. There is also a `fixprimes` tweak that we can use to scale and fix the glyph itself. Below, we see how very different the primes from different fonts look (all examples are uniformly scaled), and then examples comparing the original and tweaked primes.

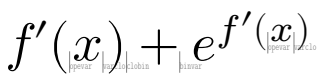
25.8 Font parameters

We add some font parameters, ignore some existing ones, and fix at runtime those that look to be suboptimal. We have no better method than looking at examples, so parameters might be fine-tuned further in the future.

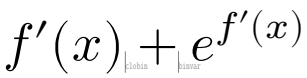
We have already mentioned that we have a few new parameters, `PrimeShiftUp` and `PrimeShiftUpCramped`, to position primes on their own axis, independent of the superscripts. They are also chosen to always be placed outside superscripts, so the inputs `$f'^2$` and `$f^2'$` both result in $f^{2\prime}$. Authors should use parentheses in order to avoid confusion.

Let us briefly mention the other parameters. These are the adapted parameters for TEX Gyre Bonum:

```
AccentTopShiftUp                    =  -15
```



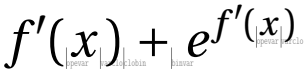
lm original



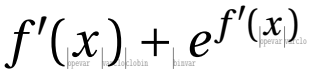
lm tweaked

lucida original

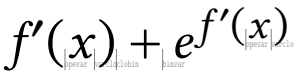
lucida tweaked



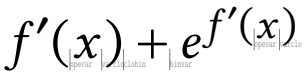
erewhon original



erewhon tweaked



libertinus original



libertinus tweaked

$$h^3 + h_2 + h_2^3 + h'$$

$$h^3 + h_2 + h_2^3 + h'$$

$$h^3 + h_2 + h_2^3 + h'$$

```
FlattenedAccentTopShiftUp         =  -15
AccentBaseDepth                   =   50
DelimiterPercent                  =   90
DelimiterShortfall                =  400
DisplayOperatorMinHeight          = 1900
SubscriptShiftDown                =  201
SuperscriptShiftUp                =  364
SubscriptShiftDownWithSuperscript = "1.4*SubscriptShiftDown"
PrimeShiftUp                      = "1.25*SuperscriptShiftUp"
PrimeShiftUpCramped               = "1.25*SuperscriptShiftUp"
```

Some of these are not in OpenType. We can set up much more, but it depends on the font what is needed, and also on user demands.

We have noticed that many font designers seem to have had problems setting some of the values; for example, `DisplayOperatorMinHeight` seems to be off in many fonts.

25.9 Profiling

Let us end with profiling, which is only indirectly related to the tweaking of the fonts. Indeed, font parameters control the vertical positioning of sub- and superscripts. If not carefully set, they might force a non-negative `\lineskip` where not necessary. In the previous section we showed how these parameters were tweaked for Bonum.

Sometimes formulas are too high (or have a too large depth) for the line, and so a `\lineskip` is added so that the lines do not clash. If the lowest part of the top line (typically caused by the depth) and the tallest

part of the bottom line (caused by the height) are not close to each other on the line, one might argue that this `\lineskip` does not have to be added, or at least with reduced amount. This is possible to achieve by adding `\setupalign[profile]`. Let us look at one example.

So the question is: how good an approximation to σ is $\sigma * W\phi$? But the attentive reader will realize that we have already answered this question in the course of proving the sharp Gårding inequality. Indeed, suppose $\phi \in \mathcal{S}$ is even and $\|\phi\|_2 = 1$, and set $\phi^a(x) = a^{n/4}\phi(a^{1/2}x)$. Then we have shown (cf. Remark (2.89)) that $\sigma - \sigma * W\phi^a \in S^{m-(\rho-\delta)}_{\rho,\delta}$ whenever $\sigma \in S^m_{\rho,\delta}$ is supported in a set where $\langle\xi\rangle^{\rho+\delta} \approx a$.

No profiling

So the question is: how good an approximation to σ is $\sigma * W\phi$? But the attentive reader will realize that we have already answered this question in the course of proving the sharp Gårding inequality. Indeed, suppose $\phi \in \mathcal{S}$ is even and $\|\phi\|_2 = 1$, and set $\phi^a(x) = a^{n/4}\phi(a^{1/2}x)$. Then we have shown (cf. Remark (2.89)) that $\sigma - \sigma * W\phi^a \in S^{m-(\rho-\delta)}_{\rho,\delta}$ whenever $\sigma \in S^m_{\rho,\delta}$ is supported in a set where $\langle\xi\rangle^{\rho+\delta} \approx a$.

Profiling

In the above paragraphs we enabled a helper that shows us where the profiling feature kicks in. We also show the lines (`\showmakeup[line]`). Below we show the example without those helpers. You can judge for yourself which one you prefer.

So the question is: how good an approximation to σ is $\sigma * W\phi$? But the attentive reader will realize that we have already answered this question in the course of proving the sharp Gårding inequality. Indeed, suppose $\phi \in \mathcal{S}$ is even and $\|\phi\|_2 = 1$, and set $\phi^a(x) = a^{n/4}\phi(a^{1/2}x)$. Then we have shown (cf. Remark (2.89)) that $\sigma - \sigma * W\phi^a \in S^{m-(\rho-\delta)}_{\rho,\delta}$ whenever $\sigma \in S^m_{\rho,\delta}$ is supported in a set where $\langle\xi\rangle^{\rho+\delta} \approx a$.

No profiling

So the question is: how good an approximation to σ is $\sigma * W\phi$? But the attentive reader will realize that we have already answered this question in the course of proving the sharp Gårding inequality. Indeed, suppose $\phi \in \mathcal{S}$ is even and $\|\phi\|_2 = 1$, and set $\phi^a(x) = a^{n/4}\phi(a^{1/2}x)$. Then we have shown (cf. Remark (2.89)) that $\sigma - \sigma * W\phi^a \in S^{m-(\rho-\delta)}_{\rho,\delta}$ whenever $\sigma \in S^m_{\rho,\delta}$ is supported in a set where $\langle\xi\rangle^{\rho+\delta} \approx a$.

Profiling

It is worth emphasizing that, contrary to what one might believe at first, the profiling does not substantially affect the compilation time. On a 300-page math book we tried, which usually compiles in about 10 seconds, profiling did not add more than 0.5 seconds. The same observation holds for the other math tweaks we have mentioned: the overhead is negligible.

25.10 Conclusions

All these tweaks can be overloaded per glyph if needed; for some fonts, we indeed do this, in so-called goodie files. The good news is that by doing all this we present the engine with a font that is consistent, which also means that we can more easily control the typeset result in specific circumstances.

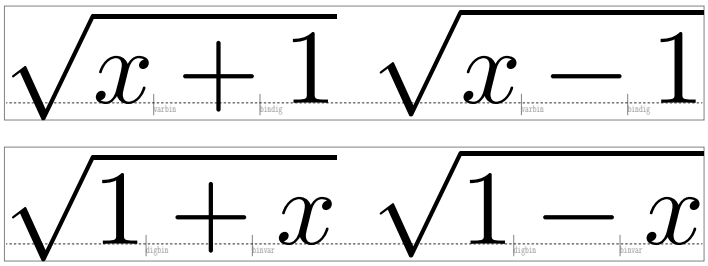
The reader may wonder how we ended up with this somewhat confusing state of affairs in the font world. Here are some possible reasons. There is only one reference font, Cambria, and that uses its reference word processor renderer, Word. Then came XeTeX that as far as we know maps OpenType math onto a traditional TeX engine, so when fonts started coming from the TeX crowd, traditional dimensions and parameters sort of fit in. When LuaTeX showed up, it started from the other end: OpenType. That works well with the reference font but less so with that ones coming from TeX. Eventually more fonts showed up, and it's not clear how

these got tested because some lean towards the traditional and others towards the reference fonts. And, all in all, these fonts mostly seem to be rather untested in real (more complex) math.

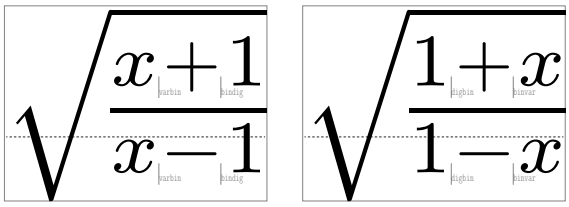
The more we looked into the specific properties of OpenType math fonts and rendering, the more we got the feeling that it was some hybrid of what TEX does (with fonts) and ultimately desired behavior. That works well with Cambria and a more or less frozen approach in a word processor, but doesn't suit well with TEX. Bits and pieces are missing, which could have been added from the perspective of generalization and imperfections in TEX as well. Lessons learned from decades of dealing with math in macros and math fonts were not reflected in the OpenType fonts and approach, which is of course understandable as OpenType math never especially aimed at TEX. But that also means that at some point one has to draw conclusions and make decisions, which is what we do in ConTEXt, LuaMetaTEX and the runtime-adapted fonts. And it gives pretty good and reliable results.

26 Somewhat radical

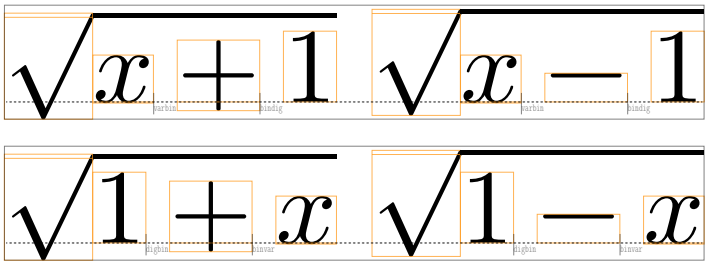
Here we will discuss an aspect of radicals, namely how variants get applied. Take the following situation:



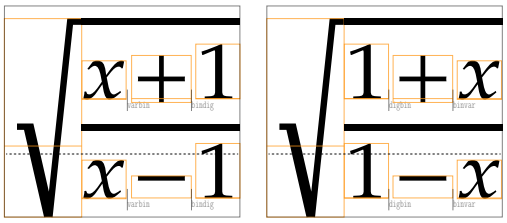
Watch the slight difference in radical heights. Now look at this:



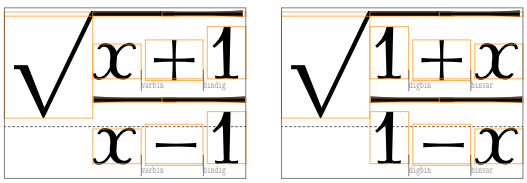
Here we need to make sure that we don't run into the slope, because, when we have a close look at the shapes we see that the radical symbol has a tight bounding box:



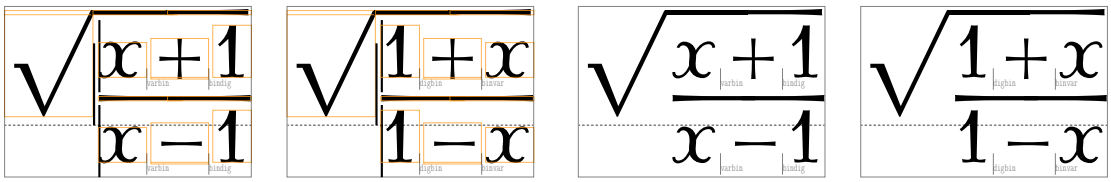
In pagella we get:



and in antykwa:



But now look at this formula:



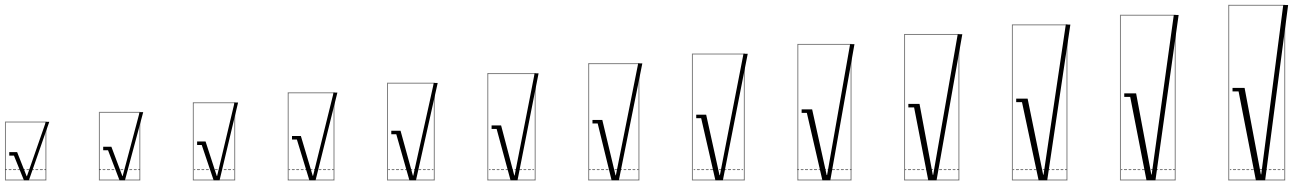
Here we see several mechanisms in action and for a good reason. First of all we want similar subformulas (under the symbol) to have compatible radicals. For this we use special struts so that we always have at least some height. We also compensate for slight differences in depth by setting a minimum depth. Finally we add a bit of margin. That last feature moves the content free from the symbols which means that we can have less distance between the top of the content and the rule. In many fonts that distance is set to a value that prevents clashes and the more slope we have, the more opportunity there is to clash.

When the best fit decision is made for a radical, the effective height of the content (height plus depth of the box) is incremented by a gap variable. The standard specifies the `RadicalVerticalGap` as "Space between the (ink) top of the expression and the bar over it. Suggested: 1.25 default rule thickness." and `RadicalDisplayStyleVerticalGap` as "Space between the (ink) top of the expression and the bar over it. Suggested: default rule thickness plus .25 times x-height.". These values are actually rather font dependent because the slope needs to be taken into account; there is also a visual aspect to it.

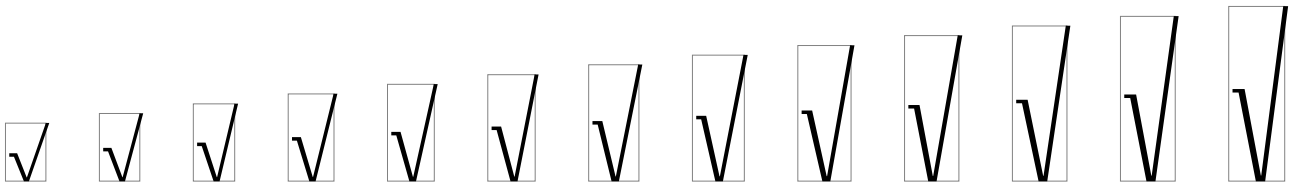
We can't tweak the radical width because the rule has to be attached. If we could we'd have to do it for every variant. So, instead we set up radical like this:

```
\setupmathradical
  [strut=height,     % only height
   leftmargin=.05mq, % fraction of math quad
   mindepth=.05mx]   % fraction of math x height
```

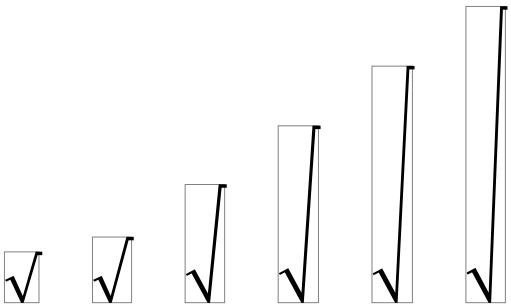
When deciding what size to use, a list of variants is followed till there is a match and when we run out of variants an extensible is constructed. Here is the list of possible sizes in the current font:



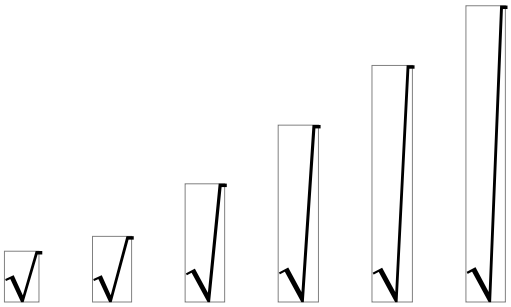
However, when we don't center around the math axis we get a more distinctive view on the steps:



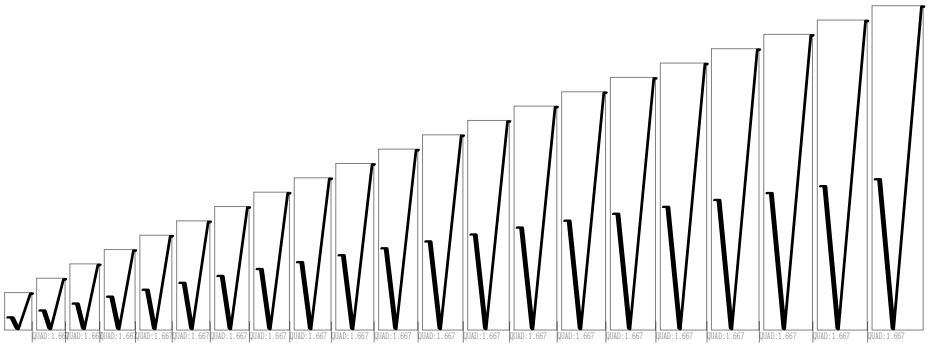
It will be clear that the steps can't be too large but there are fonts out there that behave rather extreme, like Cambria:



The only way out here is to either inject scaled variants into the list of possibilities or to simply ignore all except the first one and go straight to the extensible, so that's what we do, in combination with tweaked parameters and a margin:



As with many font parameters (also in text) one sometimes wonder if font designer test with real examples. There are of course exceptions, for instance the `ebgaramond` font, but that one goes over the top in other areas. Here one can also wonder if the upper half of the range makes sense over an extensible. For consistency one wants steps to be not too small, so that a sequence of radicals looks simular, but steps larger than for instance the height are probably bad.

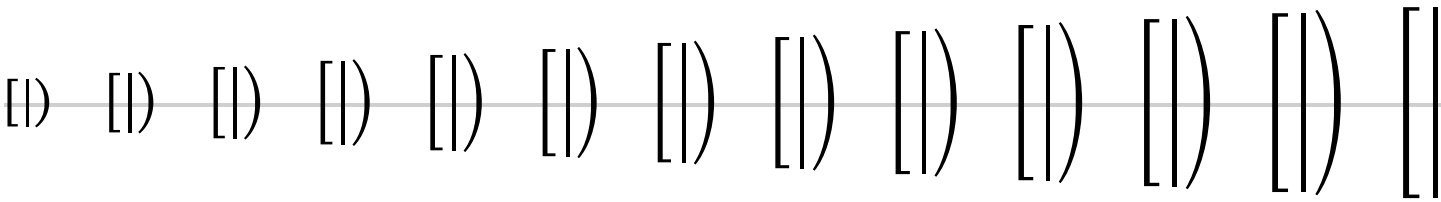


So, as with other examples that we give of tweaking math, it is clear that there is no way around also tweaking radicals, and we're not even talking of the way we fine tune the positioning of degrees in radicals because that is also a neglected area in OpenType math fonts.

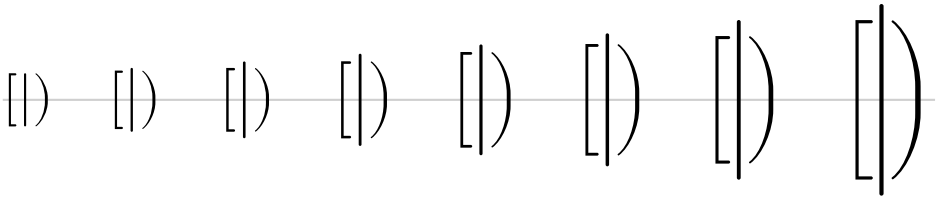
27 Between bars

Inconsistencies

The bar in math can a real pain. There are several reasons for this, for instance there is no proper left, middle and right bar in Unicode and as a result there is more work involved in getting them spaced well. Another possible issue can be to make them fit well with other fences. You expect the bars in $(x \mid y)$ and $|x \mid y|$ to look similar.



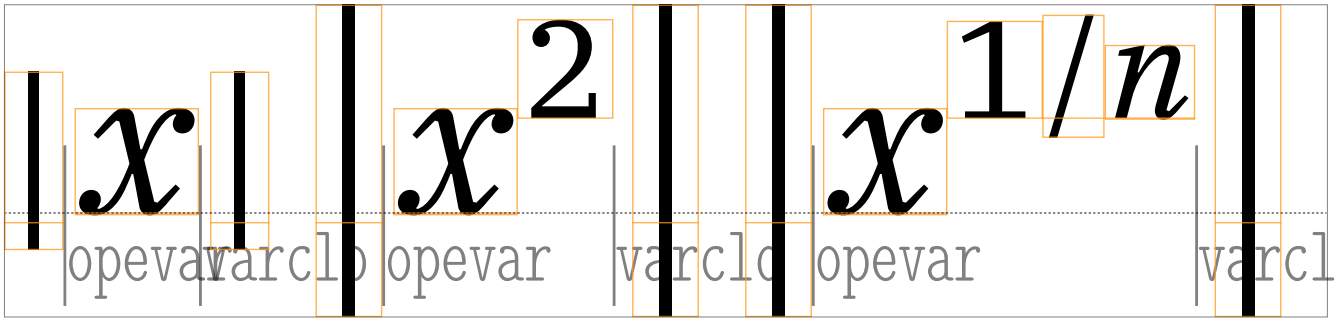
However, math fonts have their surprises:



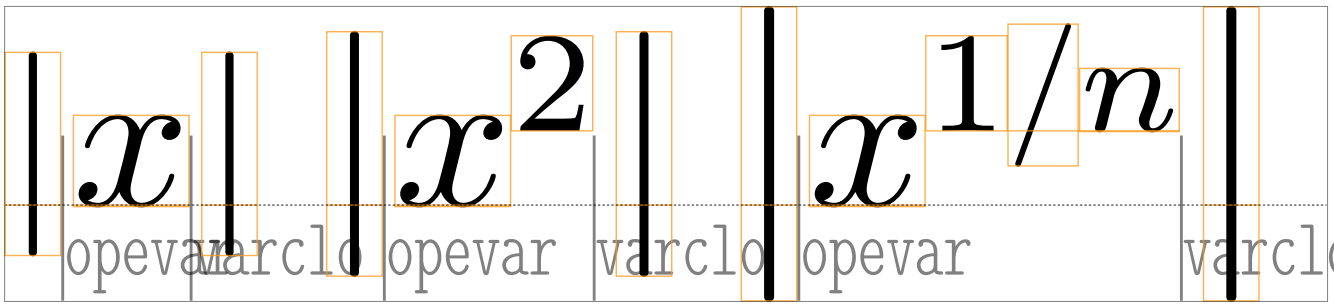
In Latin Modern only the first variant is tuned to work together but for larger sizes the bars stick out. This is a problem when we want fences to adapt. The fact that such side effects probably get unnoticed comes from the fact that macro packages assume `\bigg` and friends to be used but in ConTEXt, and especially LMTX, we have various mechanisms for this. One method is based on selecting specific variants, in the case of Latin Modern 1, 4, 6 and 7, where in in fact 7 is the last one before we switch to extensible fences. One can try to use a different selection for brackets and bars when there is no nice match but there are no equal height matches.

```
\im {\left| x       \right|}
\im {\left| x^2     \right|}
\im {\left| x^{1/n} \right|}
```

These example formulas can trigger a larger fence:

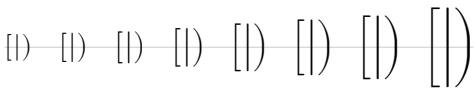


In untweaked Latin Modern we get this:

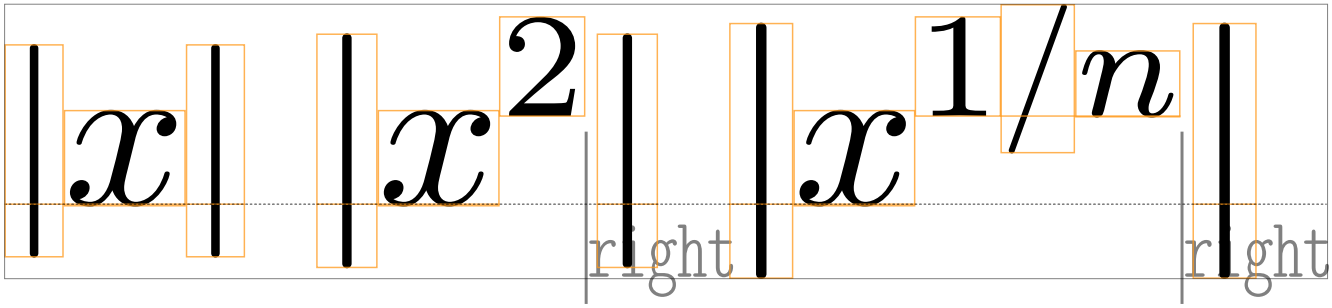


The slash in this font is rather high and therefore triggers the larger fence. One can configure this with the `\delimiterfactor` and `\delimitershortfall` but as you can see values have to relate to the font. In LMTX we set them to 1000 and 0pt and use the LuaMetaTEX equivalent font variables instead, so we can indeed fine tune per font.

To come back to the mismatch in fences, this is dealt with in a tweak: we scale the single, double and triple bars to match the brackets:



Combined with proper settings for the factor (or percentage in OpenType math speak) and shortfall, we now get:



When we let the upgraded math subsystem evolve we make many examples. Unfortunately there is always an exception. For instance, we test a specific font, notice something, deal with it, even test all fonts in inline and display math and then after months the exception shows up. In this case it was the $1/n$ in a superscript that (only?) in Latin Modern goes over the top. Actually we had noticed that bars are often inconsistent so we had a `fixbars` tweak, However, for Latin Modern we found that the inconsistency between bars and other fences needed something more drastic. Of course fixing the font is best but we're beyond that stage now: the fonts are basically frozen.

A close inspection of the too large fence which itself results from it being larger than expected by design (which we noticed by adding parentheses) itself was the result from deciding to configure additional inter-atom spacing for open and close fences (see below) which then brings us back to the fact that one bar serves three purposes. We might actually introduce these three (left, middle and right) at some point.

$$\left|\frac{n^3-2n+1}{n^5-3}-0\right|<\frac{4}{n^2}$$

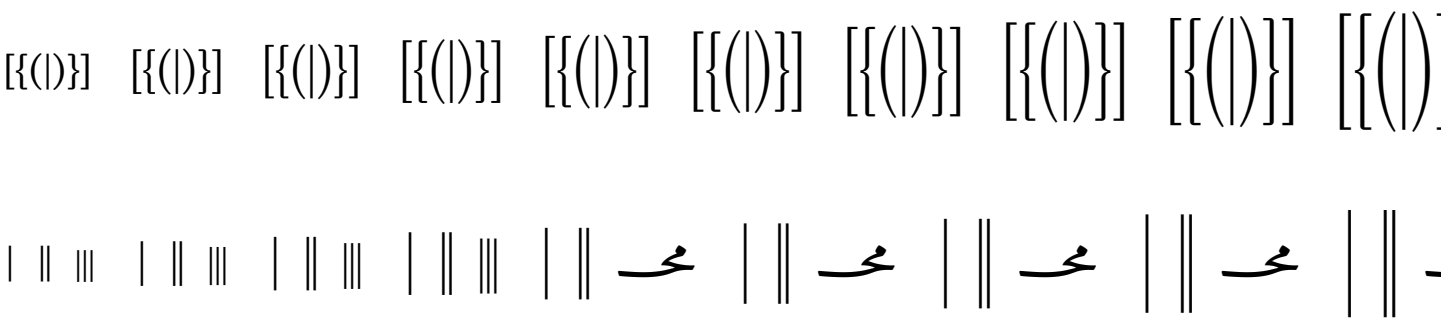
$$\left|\frac{n^3-2n+1}{n^5-3}-0\right|<\frac{4}{n^2}$$

Missing shapes

The tweak discusses in the previous section is a brute force one: we put an extensible on the base glyph. Among the arguments for doing this is that we want to be able to add consistent double and triple bars. Without mentioning fonts explicitly (as some might get fixed after we files bug reports) this is what we observed:

- There are single bars, double bars and triple bars and each has variants and extensibles. This is okay.
- Most are there but the triple bar has no variants and extensibles
- We have all three base characters but no variants. The extensible has a different width.
- Single, double and triple bars are inconsistent with each other.
- Everything is there but widths differ per variant; some match the parenthesis, brackets and braces but not consistently.
- The different variant sizes are out of sync with the sizes of parenthesis etc. and this makes for inconsistent matches, especially when also the width and positioning differs.
- Spacing between and around double and triple bars isn't always consistent.

These observations lead us to the conclusion that there is no single tweak that can fix this. Adapting the 'addbars' tweak to deal with all this made for too many alternatives in checks and fixes to feel comfortable with. This is why we decided to come up with companion fonts that provide the missing double and triple variants and extensibles consistent with the single ones, fix spacing in double and triple ones, fix inconsistent widths of bars, etc. Minor details like bad positioning are already handled well do we can keep the 'design' as it is.



Different sizes

middle

28 Getting rid of math rules

There are a few characteristics of typesetting math that determine how the internals of the TEX math engine are shaped:

- A formula is made from atoms: in $x + 1$ there are three of them. Sometimes a few are combined into subformulas.
- An atom can have scripts attached: in x_i^2 we have a super- and subscript. We can also have prescripts and indices.
- Spacing between atoms depends on what it represents: a operator, binary, ordinal etc. Sometimes we look back, sometimes we look ahead.
- The raw math list is stepwise converted to a normal node list. A next pass can use information from a previous one.
- Fractions are implemented as dedicated objects with two atoms on top of each other and optionally scripts and fences.
- Accents can be put on top of or below an atom or subformula and are an object too. Again they can have scripts.
- Fences that are supposed to scale can wrap an atom or subformula and are an object. Scripts can be attached as usual.
- Radicals have a symbols in from that extends over the atom or subformula and therefore they are objects. Such an object can have a degree and scripts.

Some specific constructs like integrals and actuarians use these mechanisms and don't have their own. So, for instance 'radical' is more a generic construct than one specific for roots.

Although in principle one can implement a math renderer in TEX it is the interplay between atoms, objects, scripts, fence sizing, spacing, etc, that makes the argument for using a hard-coded solution. Because there is some general agreement about how math should be rendered, this also results in some standardization. There are plenty parameters that drive this process and especially the extensive interaction between fonts and "what to do in what situation" makes it more convenient to do this in the engine than in macro code. There is a mix of styles, fonts, nesting, local and per-formula data management involved and not all of that is easy in macros.

Fonts play an important role in the rendering: they provide the symbols and also the specialized means for composing radicals, fences, scripts, etc. There are two aspects that have determined how the engine deals with it: the (sometimes pseudo) width and (sometimes abused) italic correction, extensible snippets that get combined into a shape of a defined size, and rules that complement these extensible snippets or stand on their own. None of the fonts that we have available is perfect and the way around that in ConTEXt is to tweak them at runtime when we load them.

Tweaks have some history. In MkIV they were mostly responsible for fixing dimensions and providing additional extensibles. In LMTX we go further and also use them to improve rendering for which they cooperate with additional engine features. During the years that we implemented new and better tweaks the mechanism in ConTEXt that deal with atoms, fractions, radicals, accents and fences have been partially redone too. Although the engine can do more, we don't always use the new features. Some are just there because

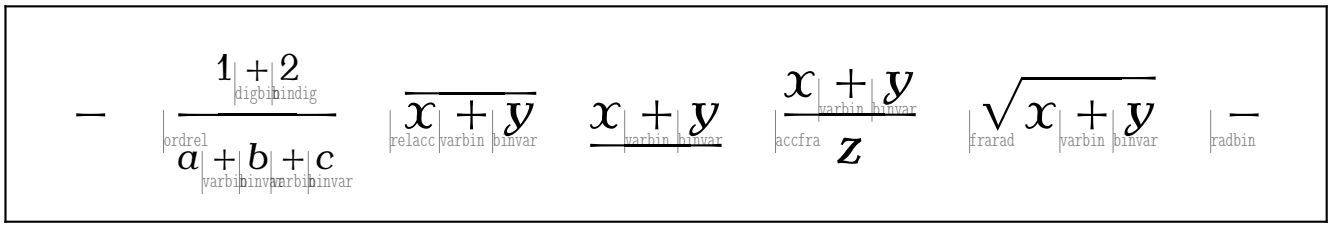
we wanted to improve the traditional approach. We can actually simplify the math engine a lot by for instance removing all the code that deals with italic correction (kind of special in traditional TEX) and just wipe features we never use.

Among the tasks that the engine delegates to macros is the construction of adaptive extensibles, especially those combined with text and most of these are horizontal: arrow or combinations of arrows with other frills, either or not with tex on top or below. In traditional fonts these are made from tips and rules, (repeated) minuses, (repeated) equals and other symbols with compensating spaces in between, to get rid of side bearings, and driven by for instance TEX's leader feature. A lot of the chosen methods has to do with the lack of slots in fonts (to create extensibles) and the fact that commands are used to inject them in formulas. One can hide a lot in a command (macro) and the user doesn't see how ugly the hacks are. But no matter what one comes up with, rules are used in combination with glyphs. Till recently.

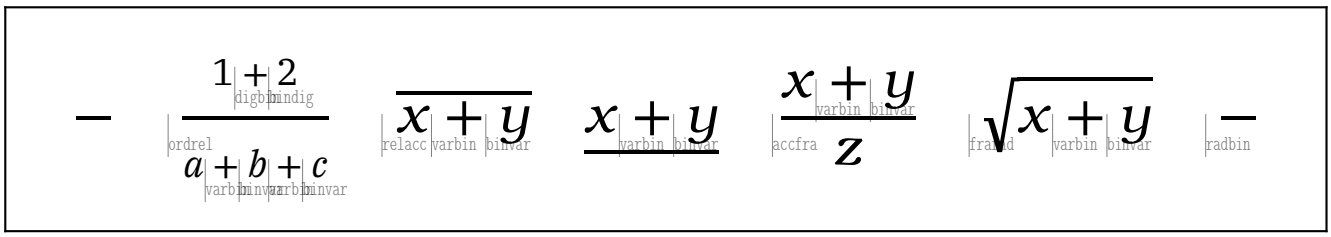
When we were making sure that most OpenType math fonts can make nice math using tweaks, we wondered what to do with the few reasonable Type1 math fonts left: the Antykwa, Iwona and Kurier. In retrospect we should have made a few companion fonts with extra glyphs but that insight came after re-implementing (read: stripping) virtual Unicode math to just suit these fonts. A nice challenge was to make "everything ruled" to use the more curved rule-based symbols in Antykwa and once that was done a logical next step was to see if that could also be done with tweaks in OpenType fonts. Once we got started with that the aim became to completely eradicate the use of rules, which are on the one hand made for some additions to the engine and on the other hand simplified related TEX code. It also introduced a few more tweaks for fonts that lack snippets that replace rules. The mechanisms that are involved are accents (normally thinner and/or smaller symbols), fences (bars), radicals (the top rule) and what we call stackers in ConTEXt: the thicker already mentioned arrow like extensibles. In fact, the smaller accent based arrows were never in TEX fonts so we cannot use them as stackers: users expect some stability in rendering after all.

A side effect of this approach is that, if we want, we can now not only strip all the code that deals with italic correction from the engine, but also all code related to rules! And we're not only talking of rendering but also all the variables that somehow determine how rules are done. In the end, when we also rid code that we don't use half the math engine can go. Of course we've added all kind of new things too so in the end the size and complexity is still comparable to the original.

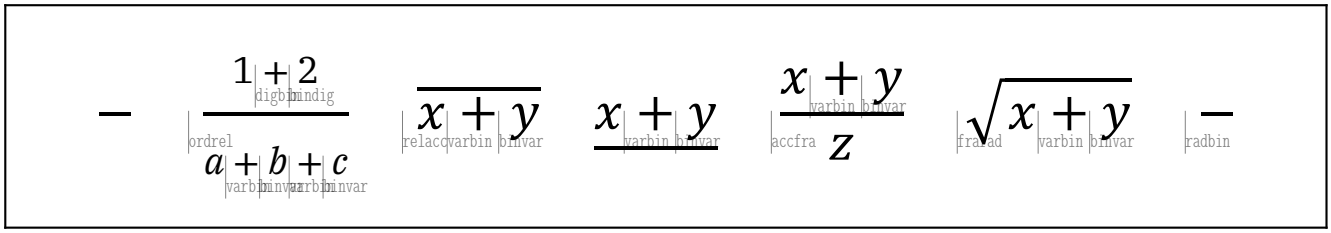
All this doesn't change the fact that the specific math representations that use some kind of 'rulish shape' is still a mess: in Unicode as well as OpenType. First of all the TEX community didn't make sure that the things we call stackers in ConTEXt got code points, which means that for instance a so called relbar is basically a stretched minus and that symbols that represent implying are tagged as left or right arrows. Compare that with the plank constant h which resulted in omission of the italic math '*h*'.. The same is true for the vertical bars where there is no distinction between left, right and middle bars. At the same time the opportunity to get it all right in OpenType fonts has been missed too. That is why we now have to tweak our way out of it.



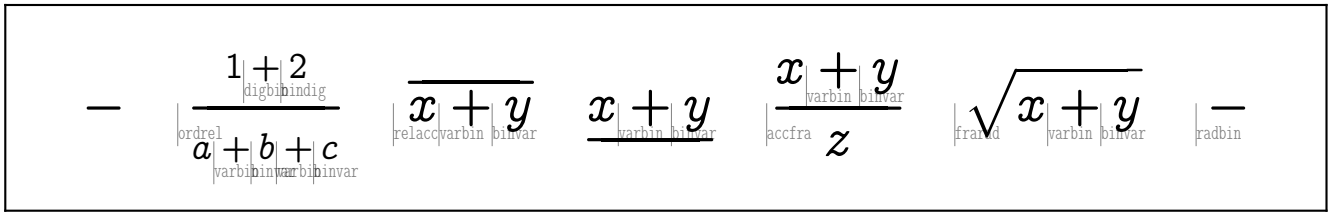
antykwa



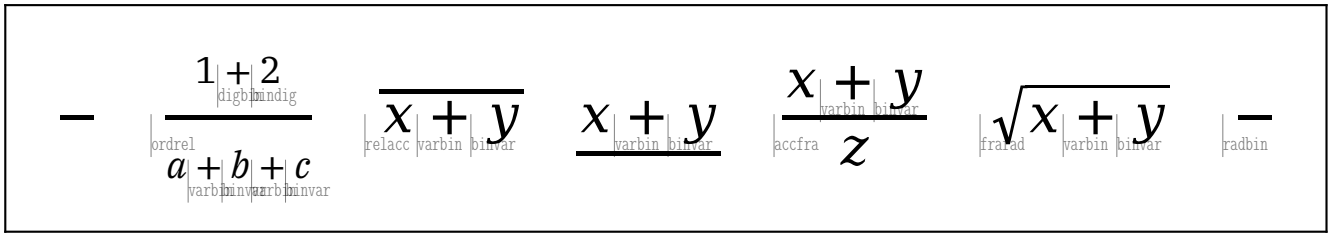
bonum



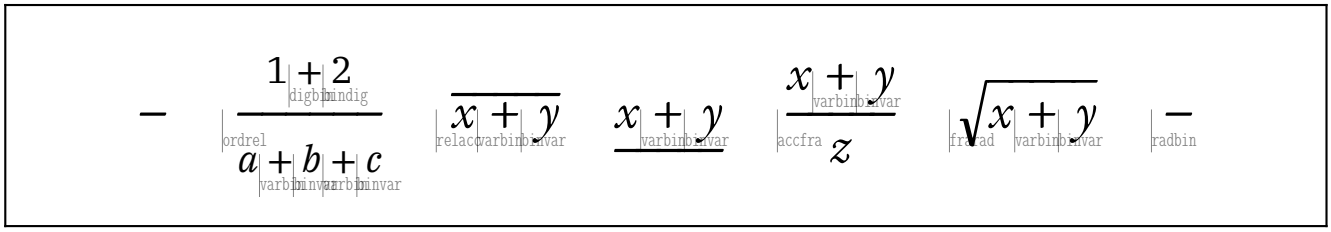
cambria



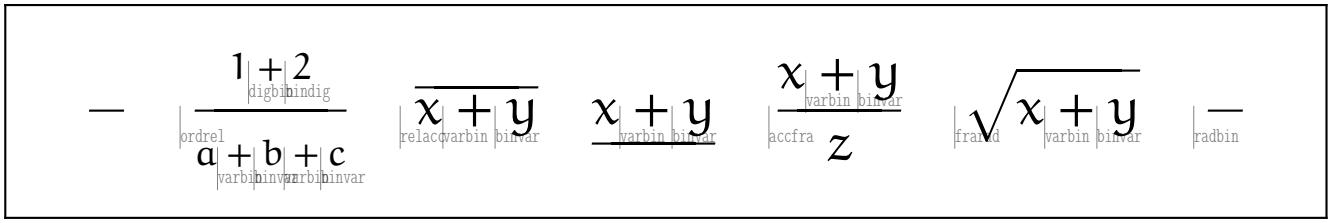
concrete



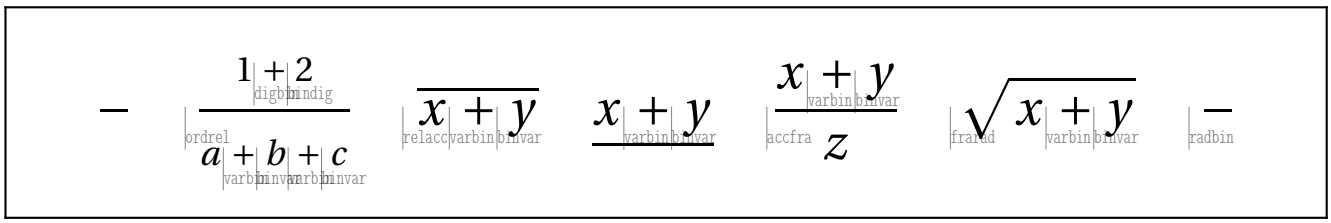
dejavu



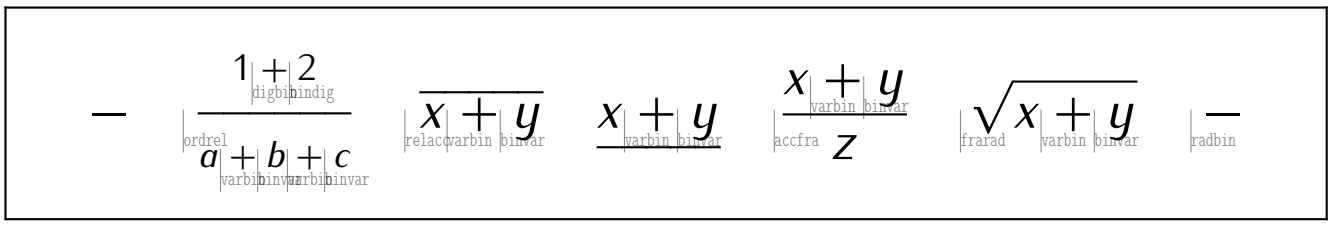
ebgaramond



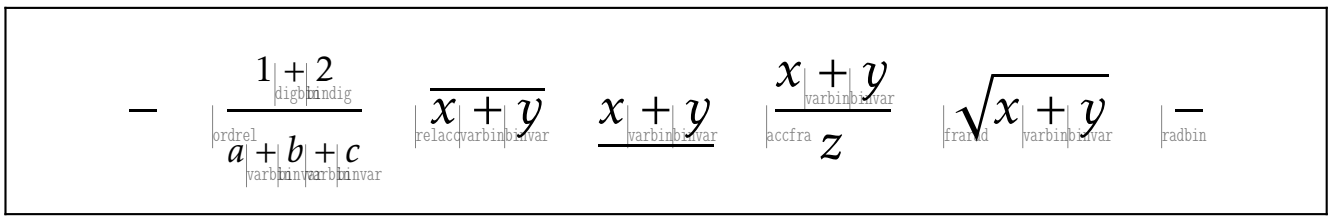
euleroverpagella



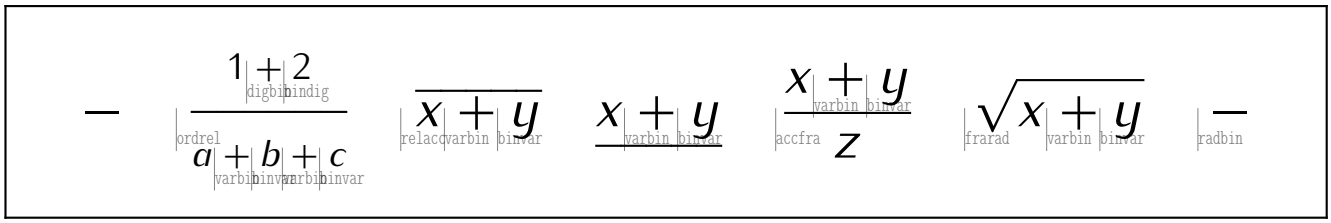
erewhon



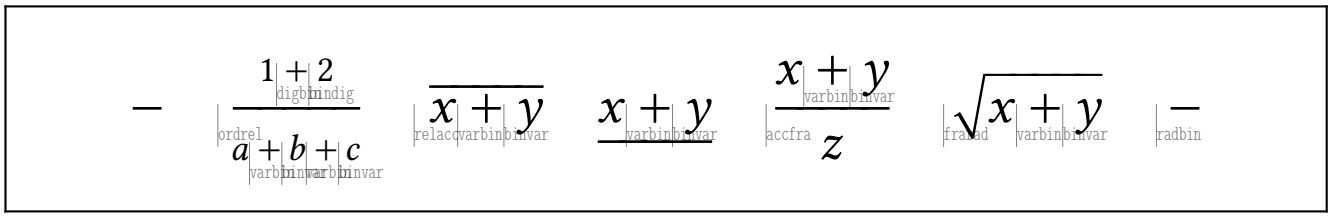
iwona



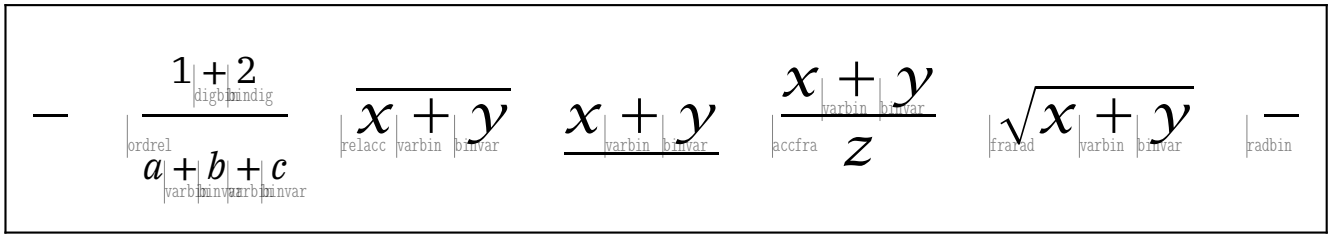
kpfonts



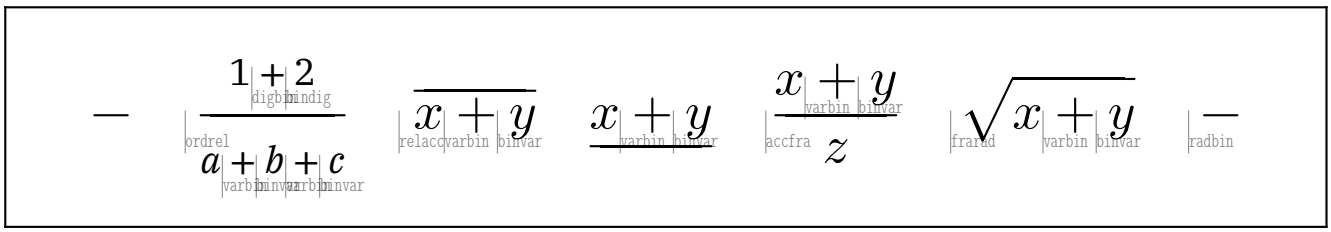
kurier



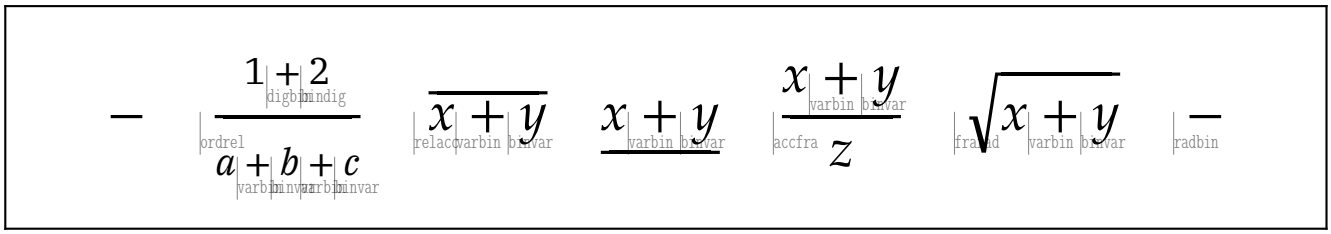
libertinus



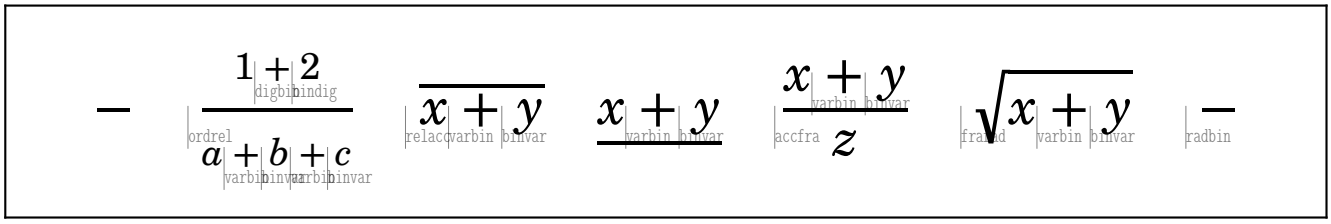
lucida



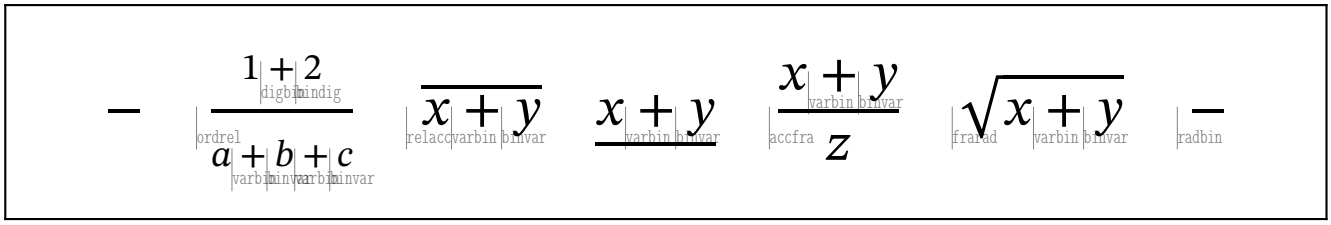
modern



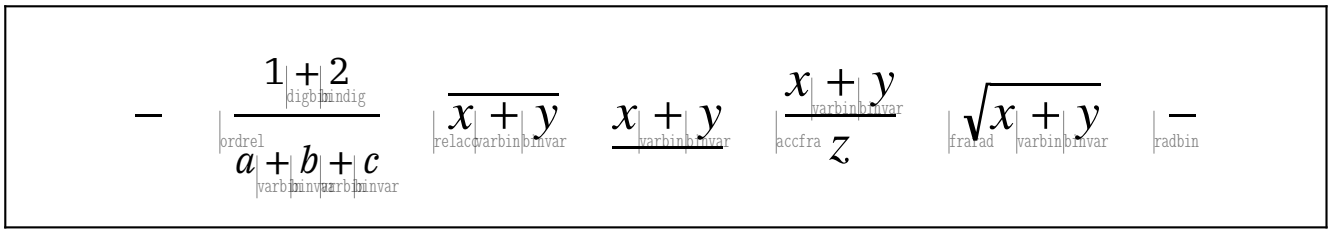
pagella



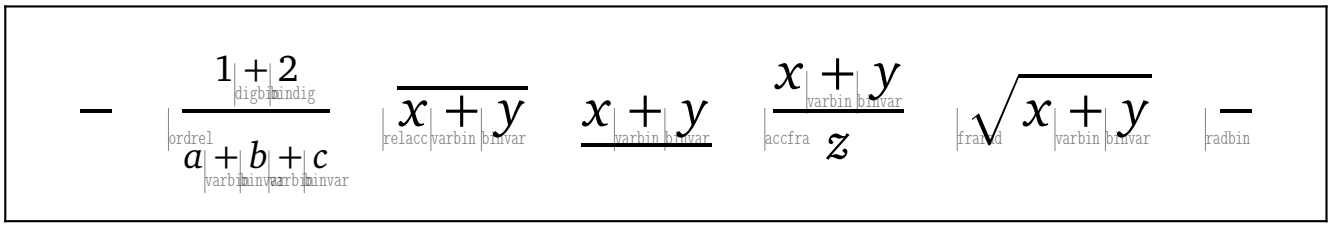
schola



stixtwo



termes



xcharter

From the perspective of the macro package (read: ConTEXt) it means that we need to provide extensibles for relbar like stackers, over- and underbars, fraction separators, and radical top elements. Where possible we can borrow snippets from the extensible minus and tweak their dimensions and when a font lacks them

we fake extensibles using the regular minus. As a reminder: that ensures that we have the proper left and right tips, that often have some curvature. In a follow up we will try to implement a more organic approach to rules (see figure 28.1).

Figure 28.1 An experiment with organic square roots in TEX.

29 Unusual bitmaps

29.1 Introduction

In the early days of TEX, fonts mostly were bitmaps and when such a bitmap was shown in zeros and ones, the shape was rather recognizable. A recent example of a special-purpose (TAOCP) bitmap font is Don Knuths three-six font. Do you recognize this character?

```
1111111111
1100110011
1100110011
0000110000
0000110000
0000110000
0001111000
0001111000
```

It has a rather low resolution, but can still serve its purpose as we will see later on. One problem is of course that such a low resolution doesn't render too well. Although vector images are often preferred, especially in a MetaPost context, below we will explain how we can also use bitmaps in a constructive way.

This article appeared in the first 2024 issue of TugBoat and as usual Karl Berry helped to make it better than it was submitted.

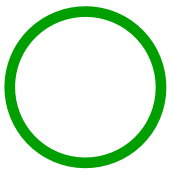
29.2 Vectorizing bitmaps

The potrace library by Peter Selinger is a nice tool: you feed it a bitmap and are rewarded with an outline specification. Details about the process can be found in https://potrace.sourceforge.net/potrace.pdf. When you limit yourself to only the basics, there are not that many source files and therefore I decided to add it to LuaMetaTEX in order to explore if we can do runtime conversions. Possible applications are logos and maybe converted bitmap fonts, although these can best be prepared beforehand because consistent metrics need to be taken care of. There are, however, other applications possible. Here I will discuss usage only in the perspective of MetaFun, simply because we have to be visual, and MetaFun is all about that. The library is of course accessible from the Lua end, if only because that way we can use it in MetaFun.

We start with a simple example that also shows a potential usage:

```
\startMPcode
    string s ; s := "010 111 010";

    draw lmt_potraced [
        bytes = s,
    ] ysized 2cm
      withpen pencircle scaled 4
      withcolor darkgreen ;
\stopMPcode
```

We feed the `lmt_potraced` macro a 3×3 bitmap encoded as a string and this is what we get back:



We expect a cross and get back a circle which is not what we want. This is because we don't have much body in this bitmap and potrace needs more pixels in order to give back a decent outline.

```
\startMPcode
    string s ; s := "010 111 010";

    draw lmt_potraced [
        bytes   = s,
        explode = true,
    ] ysized 2cm
      withpen pencircle scaled 4
      withcolor darkgreen ;
\stopMPcode
```

So, this time we 'explode' the bitmap, that is: we repeat every pixel three times in the horizontal and vertical direction. One can specify `nx` and `ny` but so far using different values doesn't help more than the magic threesome.

We're getting there but need to do a bit more.

```
\startMPcode
    string s ; s := "010 111 010";

    draw lmt_potraced [
        bytes     = s,
        threshold = 0.25,
        explode   = true,
    ] ysized 2cm
      withpen pencircle scaled 4
      withcolor darkgreen ;
\stopMPcode
```

Here we've set a threshold which will clip the paths in a range so that our case will get a better fit:

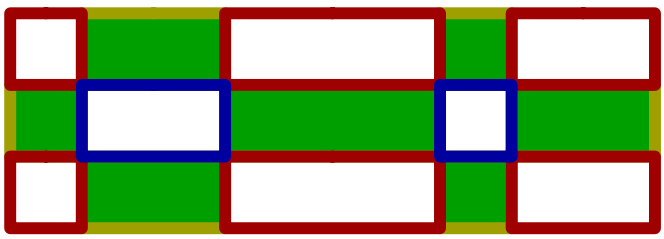
By now you will have noticed that we get back a path and this is an important feature of potrace: it returns a closed path expressed in lines and curves that one is supposed to fill. That result is converted into a Lua

table that we then can use for instance to generate a valid MetaPost path. We can do whatever we like with that path: draw or fill it for clipping. Natively, the library might return multiple paths but the user sees only one because we concatenate them which is a feature of the MetaPost library that comes with LuaMetaTEX.

Once we could do this it was no big deal to add support for filtering. After all, we have more than `0` and `1` characters available. Take this example, where we lay out the bitmap a bit differently, for clarity:

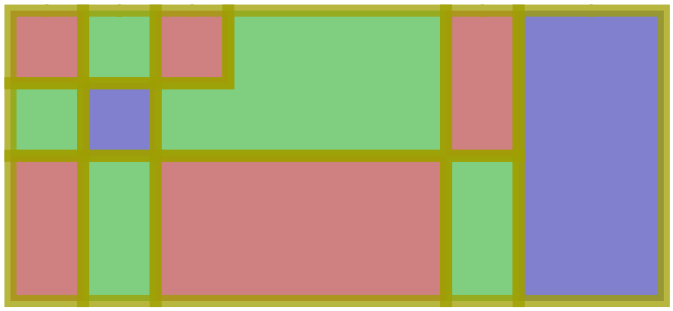
```
\startMPcode
    string s ; s := "
        211222122
        133111311
        211222122
    ";

    path p[] ;

    p[1] := lmt_potraced [
        bytes     = s,
        threshold = 0.25,
        explode   = true,
        value     = "1",
    ] ;
    p[2] := lmt_potraced [
        bytes     = s,
        threshold = 0.25,
        explode   = true,
        value     = "2",
    ] ;
    p[3] := lmt_potraced [
        bytes     = s,
        threshold = 0.25,
        explode   = true,
        value     = "3",
    ] ;

    fill p[1] withcolor darkgreen ;
    draw p[1] withcolor darkyellow ;
    draw p[2] withcolor darkred ;
    draw p[3] withcolor darkblue ;

    currentpicture := currentpicture ysized 3cm ;
\stopMPcode
```

We save the paths so that we can use them multiple times, here for a draw and fill operation on the first path but you could scale, rotate, or manipulate the result before rendering it.



This example demonstrates that a user can define outlines using a bitmap specification and that the amount of code is rather small. At some point we might add a few more helpers that might reduce the amount of code even more.

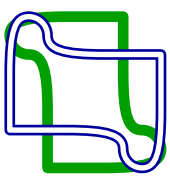
Because we have single paths we can safely apply properties like transparency, as shown below: crossing lines come out right instead of with accumulated transparent colors.



Sometimes you want to swap the rows and columns so we provide a feature for doing that:

```
\startMPcode
    string s[] ;

    s[1] := "1110 0110 0110 0111";
    s[2] := "1000 1111 1111 0001";

    draw lmt_potraced [
        bytes   = s[1],
        explode = true,
    ] ysized 2cm withcolor darkgreen withpen pencircle scaled 4 ;

    draw lmt_potraced [
        bytes   = s[2],
        explode = true,
    ] ysized 2cm withcolor darkblue withpen pencircle scaled 4 ;

    draw lmt_potraced [
        bytes   = s[1],
        explode = true,
        swap    = true,
    ] ysized 2cm withcolor white withpen pencircle scaled 2 ;
\stopMPcode
```

When one uses Lua input (as we will see later), one can do that when generating the bitmap. At any rate, this is what we get from the above:



The previous examples demonstrate that not much code is needed in order to achieve nice effects. It also illustrates that one needs to twist the mind a little and think of bitmap specifications as actually efficient

outline definitions. One could argue that such rectangular shapes are easy to program in MetaPost anyway, and going via bitmaps is kind of strange. So, let's move on to a more attractive example.

29.3 How about fonts

In order to demonstrate building fonts we need a decent bitmap font and it happens that Don Knuth's 'Font36' is a good candidate. We have discussed that one elsewhere and its usage can be found in the MetaFun `threesix` library. We happily borrow the definitions from that library:

```
\startMPdefinitions
string dekthreesix[] ; path shapes[] ;

def DEK(expr n, b) =
    dekthreesix[utfnum(n)] := b ;
enddef ;

DEK("0", "00111100 01111110 11000011 11000011 11000011 11000011 01111110 00111100" ) ;
DEK("1", "00011100 11111100 11101100 00001100 00001100 00001100 11111111 11111111" ) ;
DEK("2", "00111110 01110011 01110011 00000011 00001110 00111000 11110001 11111111" ) ;
DEK("3", "01111110 01100110 00000110 00111100 00000110 11100111 11100111 01111110" ) ;
DEK("4", "00001110 00011110 00110110 01100110 11111111 11111111 00000110 00001110" ) ;
DEK("5", "01111110 01111110 01000000 01111100 01000111 00000011 11000111 11111110" ) ;
DEK("6", "01111110 01100110 11000000 11011100 11100110 11000011 01100011 01111110" ) ;
DEK("7", "11111111 11111111 10000111 10001110 00011100 00011100 00011100 00011100" ) ;
DEK("8", "01111110 01100110 01100110 00111100 11000011 11000011 11000011 01111110" ) ;
DEK("9", "01111110 11000110 11000011 01100111 00111011 00000011 01100110 01111110" ) ;
DEK("A", "000110000 000111000 001111100 001101100 011001110 011111110 010000110 111100111") ;
DEK("B", "1111100 1110110 0110110 0111100 0110011 0110011 1110011 1111100") ;
DEK("C", "01111101 11110011 11100001 11100001 11100000 11100000 11100001 01111110") ;
DEK("D", "11111100 11100010 01100011 01100011 01100011 01100011 11100010 11111100") ;
DEK("E", "1111111 1110001 0110101 0111100 0110100 0110001 1110001 1111111") ;
DEK("F", "11111111 11100001 01100101 01111100 01100100 01100000 11111000 11111000") ;
DEK("G", "01111010 11100110 11100010 11100000 11100111 11100010 11100010 01111100") ;
DEK("H", "11100111 11100111 01000010 01111110 01000010 01000010 11100111 11100111") ;
DEK("I", "1111111 1111111 0011000 0011000 0011000 0011000 1111111 1111111") ;
DEK("J", "01111111 01111111 00000110 00000110 11110110 01100110 01100110 00111100") ;
DEK("K", "11101110 11100100 01101000 01110000 01111000 01101100 11100110 11101111") ;
DEK("L", "111111000 111111000 011000000 011000000 011000000 011000011 111000011 111111111") ;
DEK("M", "1100000011 1110000111 0111111110 0100110010 0100110010 0100000010 1100000011 1100000011") ;
DEK("N", "11000011 11100011 01110010 01111010 01011110 01001110 11100110 11100010") ;
DEK("O", "01111110 11000011 11000011 11000011 11000011 11000011 11000011 01111110") ;
DEK("P", "11111110 11100011 01100011 01111110 01100000 01100000 11111000 11111000") ;
DEK("Q", "01111100 11000110 11000110 11000110 11000110 11001110 11000110 01111101") ;
DEK("R", "11111100 11100110 01100110 01111100 01101000 01100100 11100010 11100111") ;
DEK("S", "01111110 11100001 11100001 01111000 00011110 10000111 11000011 10111110") ;
DEK("T", "1111111111 1100110011 1100110011 0000110000 0000110000 0000110000 0001111000 0001111000") ;
DEK("U", "111101111 111101111 011000010 011000010 011000010 011000010 011000010 001111100") ;
DEK("V", "11111000111 11111000111 01110000010 00110000100 00111000100 00011001000 00001101000
                                                                                    00001110000") ;
DEK("W", "111000000111 111000000111 110000000010 011000000100 011001000100 001101101000 001101101000
                                                                                  000110110000") ;
DEK("X", "1111001110 1111000110 0001101000 0000110000 0000110000 0001011000 0110001111 0111001111") ;
DEK("Y", "111100011 111100011 011000010 001110100 000111000 000011000 001111110 001111110") ;
DEK("Z", "11111111 10000111 00001110 00011100 00111000 01110000 11100001 11111111") ;
\stopMPdefinitions
```

We use these definitions in the MetaPost code below. Helpers like `utfnum` are part of LuaMetaFun and we use (named) colors defined at the ConTEXt end.

```
\startMPcode
```

```
def shapethem(expr first, last) =
    for i = utfnum(first) upto utfnum(last) :
        shapes[i] := lmt_potraced [
            bytes   = dekthreesix[i],
            explode = true,
          % threshold = .25, % more rectangular
            value   = "1",
        ] ;
    endfor ;
enddef ;

def drawthem(expr first, last, dx, dy) =
    numeric d ; d := 0 ;
    numeric f ; f := utfnum(first) ;
    numeric l ; l := utfnum(last) ;
    for i = f upto l :
        fill shapes[i] shifted (d,dy) withcolor "middlegray" ;
        draw shapes[i] shifted (d,dy) withcolor "darkgreen" ;
      % draw boundingbox shapes[i] shifted (d,dy);
        d := d + bbwidth(shapes[i]) + dx;
    endfor ;
enddef ;

shapethem("0","9") ;
shapethem("A","Z") ;

drawthem("0", "9", 10, -00) ;
drawthem("A", "J", 10, -30) ;
drawthem("K", "S", 10, -60) ;
drawthem("T", "Z", 10, -90) ;

currentpicture := currentpicture xsized TextWidth ;
\stopMPcode
```

Here we don't integrate it as a font but just show the characters as they come out, which hopefully is easier to understand. You can take a close look at the bitmaps in order to see what we get rendered in figure 29.1. Setting a lower threshold will give more rectangular results.

Figure 29.1 A fancy outline font

Because we're not going to use this font for typesetting here, a simple example of a definition will do. We first load an already existing module so that we don't need to define the bitmap definitions; basically they are the same as above.

```
\useMPlibrary[threesix]

\startMPcalculation{simplefun}
    vardef ThreeSixPotraced (expr code, spread, lift) =
        draw lmt_potraced [
            bytes   = code,
            explode = true,
            value   = "1",
        ] scaled (1/3) shifted (spread,lift) ;
    enddef ;
\stopMPcalculation
```

Next we register a MetaFun font using similar trickery as in that module. There we define a few variants but that is not needed here.

```
\startluacode
    local utfbyte = utf.byte

    local f_code = string.formatters['ThreeSixPotraced("%s",%s,%s);']

    function MP.registerthreesixpotraced(name)
        fonts.dropins.registerglyphs {
            name     = name,
            units    = 12,
            usecolor = true,
        }
        for u, data in table.sortedhash(MP.font36) do
            local ny     = 8
            local nx     = ((#data + 1) // ny) - 1
            local height = ny * 1.1 - 0.1
            local width  = nx * 1.1 - 0.1
            local spread = 0.9
            local lift   = 0.3
            fonts.dropins.registerglyph {
                category = name,
                unicode  = utfbyte(u),
                width    = width + spread,
                height   = height,
                code     = f_code(data,spread,lift),
            }
        end
    end

    MP.registerthreesixpotraced("fontthreesixpotraced")
\stopluacode
```

Finally we define a font feature that will hook the previous code into the font handler. There a Type3 font will be constructed.

```
\definefontfeature
  [fontthreesixpotraced]
  [default]
  [metapost=fontthreesixpotraced,
   spacing=.375 plus .2 minus .1 extra .375]

\definefont[DEKFontP][Serif*fontthreesixpotraced]
```

We now show an example of usage (abridged). This font only has uppercase characters so one might consider duplicating these into the lowercase slots. Because we replace glyphs in the serif font used, we still have a complete font, albeit of mixed design.

```
\DEKFontP \WORD{\samplefile{knuth}}
```

This gives us a rendering that is quite readable, especially when you consider how small the bitmaps are (the red bar is the overfull box marker):

THUS, I CAME TO THE CONCLUSION THAT THE DESIGNER OF A NEW SYSTEM MUST NOT ONLY BE THE IMPLEMENTER AND FIRST LARGE-SCALE USER; THE DESIGNER SHOULD ALSO WRITE THE FIRST USER MANUAL.

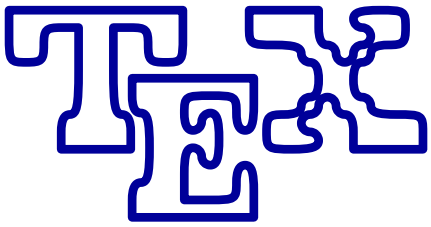
THE SEPARATION OF ANY OF THESE FOUR COMPONENTS WOULD HAVE HURT TEX SIGNIFICANTLY. IF I HAD NOT PARTICIPATED FULLY IN ALL THESE ACTIVITIES, LITERALLY HUNDREDS OF IMPROVEMENTS WOULD NEVER HAVE BEEN MADE, BECAUSE I WOULD NEVER HAVE THOUGHT OF THEM OR PERCEIVED WHY THEY WERE IMPORTANT.

BUT A SYSTEM CANNOT BE SUCCESSFUL IF IT IS TOO STRONGLY INFLUENCED BY A SINGLE PERSON. ONCE THE INITIAL DESIGN IS COMPLETE AND FAIRLY ROBUST, THE REAL TEST BEGINS AS PEOPLE WITH MANY DIFFERENT VIEWPOINTS UNDERTAKE THEIR OWN EXPERIMENTS.

Just in case Don Knuth runs into this example, we need to cheat a little here and redefine the TEX logo definition:

```
\protected\def\TeX
  {\dontleavehmode
   \begingroup
   T%
   \kern-.40\fontcharwd\font`T%
   \lower.45\fontcharht\font`X\hbox{E}%
   \kern-.15\fontcharwd\font`X%
   X%
   \endgroup}
```

A real font would have proper font kerns but if needed you can set that up using the OpenType feature plug in mechanism that ConTEXt provides. A font like this can also be colored:



29.4 External bitmaps

A convenient way to include traced images is to use the potrace command line tool. However, we can also include grayscale single-byte png-encoded images:

```
\startMPcode
    path p ; p := lmt_potraced [
        filename  = "mill.png",
        criterium = 100,
    ] ;

    fill last_potraced_bounds withcolor "middlegray" ;
    fill p withcolor "darkgreen" ;
    draw p withcolor "darkred" withpen pencircle scaled 1.5 ;
    setbounds currentpicture to last_potraced_bounds ;

    currentpicture := currentpicture ysized 10cm ;
\stopMPcode
```

This is not the best image but the photograph happens to be part of the ConTEXt distribution so why not use it. You can of course apply a different criterion and draw the subsequent trace in a different color. The result is shown in figure 29.2.

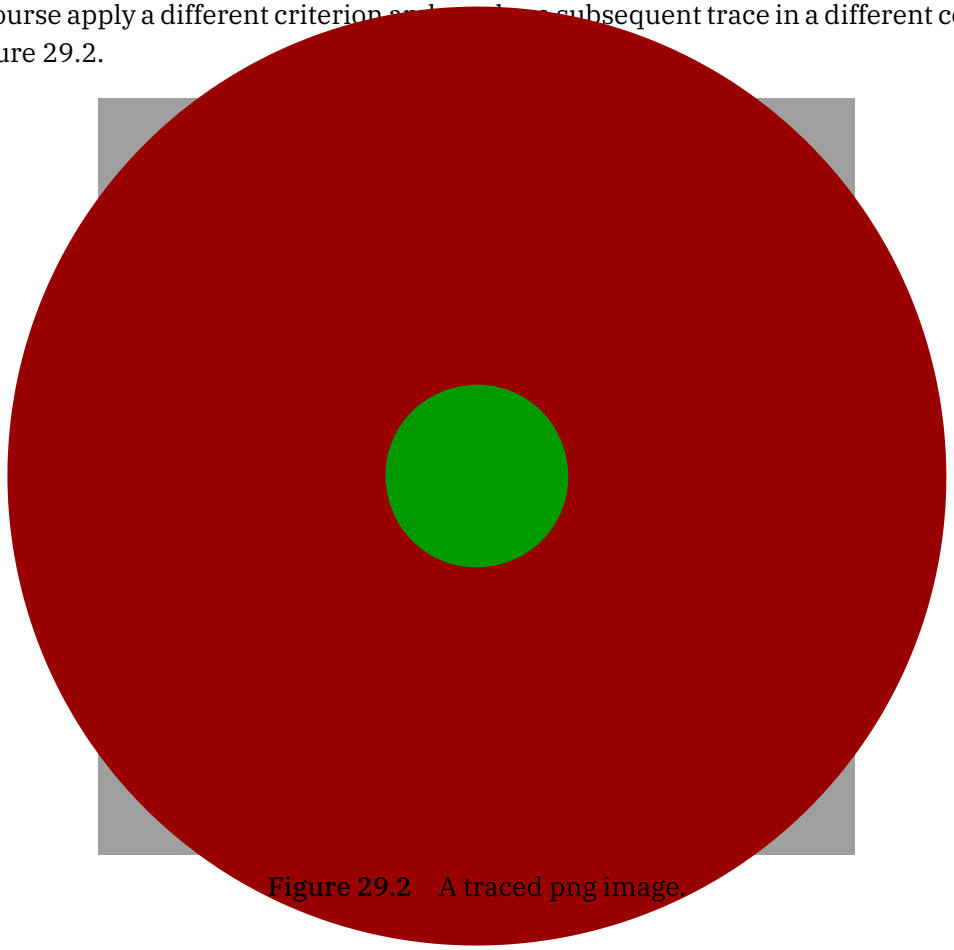


Figure 29.2 A traced png image.

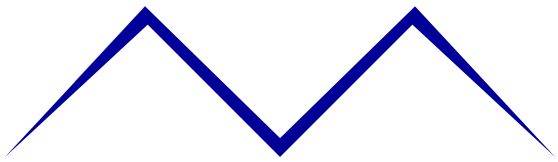
In addition to the last_potraced_bounds path variable we also have last_potraced_width and last_potraced_height numeric available.

29.5 Using LUA-generated bitmaps

It is tempting to see what can be done with a bitmap generated by Lua, but let's first give a simple example. Here we register a bitmap:

```
\startluacode
local s = [[
    00000001000000000000010000000
    00000010100000000000101000000
    00000100010000000001000100000
    00001000001000000010000010000
    00010000000100000100000001000
    00100000000010000100000000100
    01000000000001001000000000010
    10000000000001100000000000001
]]

potrace.setbitmap("mybitmap",s)
\stopluacode

\startMPcode
    path p ; p := lmt_potraced [
        stringname = "mybitmap",
    ] ;

    fill p ysized 2cm withcolor "darkblue" ;
\stopMPcode
```

We could play with the parameters but what we get is an outline that is supposed to be filled:

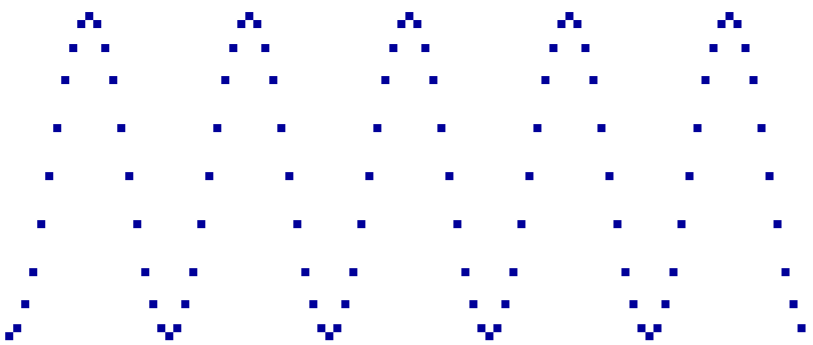


Next we create a bitmap from a dataset; in this case, we draw a function. Of course we could create some handy helpers to do this:

```
\startluacode
local d = table.setmetatableindex("table")

local t    = { }
local step = 100
local ymin = 0
local ymax = 0

local x    = 0
local dx   = 10*math.pi/step
```

```
for i=1,step do
    local y = math.round(math.cos(x)*20)
    x = x + dx
    if y > ymax then
        ymax = y
    end
    if y < ymin then
        ymin = y
    end
    t[i] = y
end

for i=1,step do
    for j=ymin, ymax do
        d[j][i] = '0'
    end
end

for i=1,step do
    d[t[i]][i] = '1'
end

for y=ymin,ymax do
    d[y] = table.concat(d[y])
end

potrace.setbitmap("mybitmap",table.concat(d," ",ymin,ymax))
\stopluacode

\startMPcode
    path p ; p := lmt_potraced [
        stringname = "mybitmap",
        explode   = true,
      % tolerance = 0.5,
        threshold = 0,
      % optimize  = true,
    ] ;

    fill p withcolor "darkblue" ;
\stopMPcode
```

To what extent this is a useful application is to be decided:



Later we will see that it is less work if we stay in the first quadrant, using $(1,1)$ as lower left corner. Here we explicitly need to provide `concat` the range because we have a negative $y_{\min}$.

It quickly gets more interesting when we use an intermediate bitmap for (a kind of) surface plots.

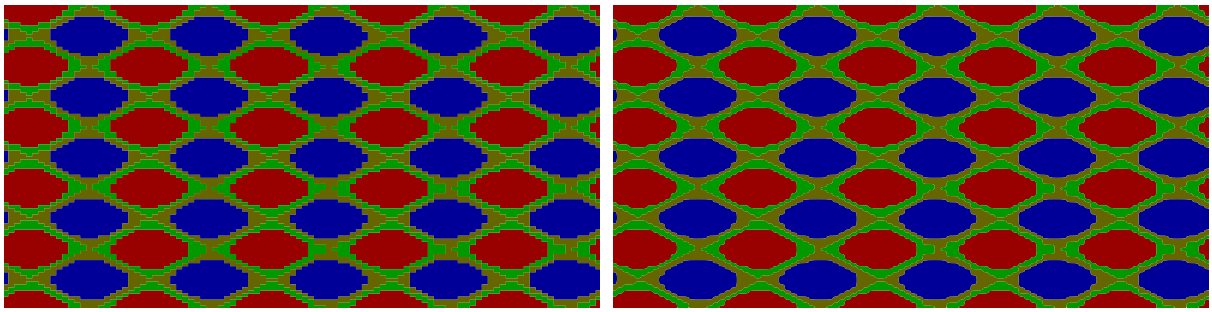
```
\startluacode
local sin, cos, pi = math.sin, math.cos, math.pi

local pp = pi/10

local function f(x,y)
    local z = sin(pp*x) + cos(pp*y)
    if z > 0.5 then
        return '1'
    elseif z > 0 then
        return '2'
    elseif z < -0.5 then
        return '3'
    else
        return'4'
    end
end

potrace.setbitmap("mybitmap",potrace.contourplot(100,100,f))
\stopluacode
```

Here we use a helper that runs the given function over the maxima and collects the results in a bitmap. More such helpers will be provided when users come up with more demands.

```
\startMPcode
    fill lmt_potraced [
        stringname = "mybitmap",
        value      = "1",
        explode    = true,
        threshold  = 0.25,
      % tolerance  = 0.1,
      % threshold  = 1.0,
        optimize   = true,
    ] withcolor "darkred" ;

    fill lmt_potraced [
        stringname = "mybitmap",
        value      = "2",
        explode    = true,
        threshold  = 0.25,
      % tolerance  = 0.1,
      % threshold  = 1.0,
        optimize   = true,
    ] withcolor "darkgreen" ;

    fill lmt_potraced [
```

```
        stringname = "mybitmap",
        value      = "3",
        explode    = true,
        threshold  = 0.25,
      % tolerance  = 0.1,
      % threshold  = 1.0,
        optimize   = true,
    ] withcolor "darkblue" ;

    fill lmt_potraced [
        stringname = "mybitmap",
        value      = "4",
        explode    = true,
        threshold  = 0.25,
      % tolerance  = 0.1,
      % threshold  = 1.0,
        optimize   = true,
    ] withcolor "darkyellow" ;

    clip currentpicture to last_potraced_bounds enlarged -1;
    currentpicture := currentpicture xysized (.45*TextWidth,4cm) ;
\stopMPcode
```

Here we are only exploring the possibilities so there is no interface like we have with contour plots. One can imagine that we provide this as a plugin, which is not that hard but whether it eventually happens depends on user demand or rainy days. So to summarize: here we generate the bitmap, we call out to potrace for a vector representation, that gets fed into MetaPost and which gives us back a result that can be converted to pdf. Of course we could go directly from potrace output to pdf if we want to, but now we get full control over the final result. In case you wonder about performance: it compiles real fast!



Some work is needed to scale the image to the proportions that reflect the input ranges but because we have a vector image that does not affect the quality of the outcome. Here we also show the effects of `tolerance` which influences the optimizing and `threshold` that determines the accuracy (number of points). In the second rendering: we use the commented values that work quite well with this kind of more mathematical images.

29.6 Experimenting

You can use bitmaps as a design tool but it needs a little experimenting to get the idea. Take these examples:

We started with a simple X-like symbol:

```
\startMPcode
    fill lmt_potraced [ bytes = "
        00100000001
        01010000010
        10001000100
        01000101000
        00100010000
        00010101000
        00001000100
        00010100010
        00100010001
        01000001010
        10000000100
    " ] xsized 2cm
        withcolor darkgreen ;
\stopMPcode
```

Next we started filling the shape a bit and tried to make it less spiked:

```
\startMPcode
    fill lmt_potraced [ bytes = "
        00100000001
        01010000011
        10001000100
        01000101000
        00100010000
        00010101000
        00001000100
        00010100010
        00100010001
        11000001010
        10000000100
    " ] xsized 2cm
        withcolor darkblue ;
\stopMPcode
```

And finally we add even more ones to the bitmap. You need to fill a bitmap area in order to get an efficient fill.

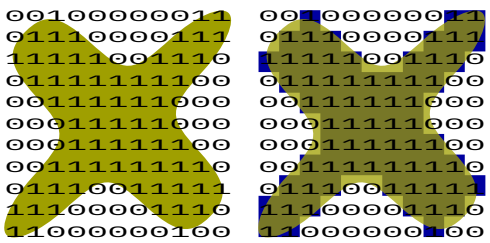
```
\startMPcode
    fill lmt_potraced [ bytes = "
        00100000011
        01110000111
        11111001110
```

```
        01111111100
        00111111000
        00011111000
        00011111100
        00111111110
        01110011111
        11100001110
        11000000100
    " ] xsized 2cm
        withcolor darkyellow ;
\stopMPcode
```

If you're in doubt you can also render the bitmap, assuming that it has reasonable proportions.

```
\startMPcode
    string s ; s := "
        00100000011
        01110000111
        11111001110
        01111111100
        00111111000
        00011111000
        00011111100
        00111111110
        01110011111
        11100001110
        11000000100
    " ;
    fill lmt_potraced [ bytes = s ]
        xsized 3cm
        withcolor darkyellow ;
    draw lmt_potraced [ bytes = s, alternative = "text" ]
        shifted (.25,.25)
        xsized 3cm ;
\stopMPcode
```

The article that we mentioned in the introduction explains how potrace looks at a bits in relation to its neighbors.

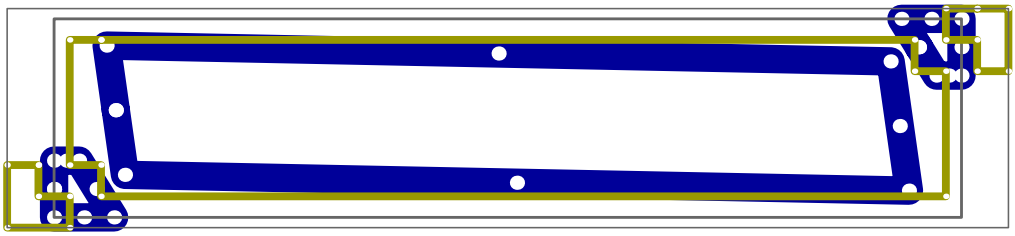


You can save some runtime (and coding) by using the start-stop wrappers that keep the (intermediate) potrace object available. This permits for instance showing the 'original' polygon that serves as basis for successive steps in the library towards to the final curve(s).

```
\startMPcode
```

```
string s ; s :=
    "01111111111111111111111111111100
     11000000000000000000000000000110
     11000000000000000000000000000011
     11000000000000000000000000000011
     11000000000000000000000000000011
     01100000000000000000000000000011
     00111111111111111111111111111110";

lmt_startpotraced [ bytes = s ] ;

    p := lmt_potraced [
        value     = "0",
        threshold = 0,
        tolerance = 0,
        optimize  = true,
    ] ;

    draw image (
        p := p shifted - center p ;
        draw p
            withpen pencircle scaled 1
            withcolor "darkblue" ;
        drawpoints p
            withpen pencircle scaled .5
            withcolor "white" ;
        draw boundingbox p
            withpen pencircle scaled .1
            withcolor "darkgray" ;
    ) ysized 3cm ;

    path p ; p := lmt_potraced [
        value   = "0",
        polygon = true,
    ] ;

    draw image (
        p := p shifted - center p ;
        draw p
            withpen pencircle scaled .25
            withcolor "middleyellow" ;
        drawpoints p
            withpen pencircle scaled .175
            withcolor "white" ;
        draw boundingbox p
            withpen pencircle scaled .05
            withcolor "darkgray" ;
    ) ysized 3cm ;

lmt_stoppotraced ;
```

```
\stopMPcode
```



The above is just a bit of exploring the possibilities so eventually there will be a chapter on this in the LuaMetaFun manual, because it is definitely fun to play with this in the perspective of MetaPost.

29.7 Contour plots

We end with showing how we can do rather nice contour plots and region plots with help of the potracer. Let us start with the latter type of graphics. In a recent math paper Mikael was counting nodal domains of Neumann eigenfunctions to the Laplace operator in a square. These eigenfunctions are built from cosines. One example is given by

$$\Psi(x,y) = \cos(8\pi x)\cos(3\pi y) + \cos(3\pi y)\cos(8\pi x).$$

The related graphic of interest is to fill the part of the unit square where Ψ is positive. In the article, Wolfram Mathematica was used to produce this graphic, with its built-in function `RegionPlot`. The result was indeed satisfactory (Figure 29.3(a)).

If one looks closely at the graphics one will see that the filled regions are made up of a mesh. With potrace we instead receive one (!) path. To generate the corresponding graphic we first use Lua to define a function that takes a point as input and returns 1 if Ψ is positive at the point and 0 otherwise. We then use it to generate a 1000×1000 bitmap image of zeros and ones accordingly.

```
\startluacode
    local cos, pi = math.cos, math.pi

    local N       = 1000
    local pp      = pi/N
    local pp3     = 3 * pp
    local pp8     = 8 * pp
    local cospp8y = 0
    local cospp3y = 0

    local function f(x,y)
        if x == 1 then
            cospp8y = cos(pp8*y)
            cospp3y = cos(pp3*y)
        end
        local z = cos(pp8*x)*cospp3y + cos(pp3*x)*cospp8y
        if z > 0 then
            return '1'
        else
            return '0'
        end
```

```
    end

    potrace.setbitmap("mybitmap", potrace.contourplot(N,N,f))
\stopluacode
```

Once this is done, we can use the result in `lmt_potraced`.

```
\startMPcode
    path p ; p := lmt_potraced [
        stringname = "mybitmap",
        value      = "1",
        threshold  = 0.25,
        optimize   = true,
    ] ;

    p := p xsized .45TextWidth ;
    fill p withcolor "darkred" ;
\stopMPcode
```

This results in Figure 29.3(b), which looks very similar to the one generated by Wolfram Mathematica. Note that `p` is one (disconnected) path.

(a) (b)

Figure 29.3 (a) A region plot generated by Wolfram Mathematica. (b) The same, generated by potrace and MetaFun.

We give one example of a contour plot. By a contour plot, here we mean a plot of the curve that is described as the solution to an equation $F(x,y)=0$. With $F(x,y)=y-f(x)$ we realize that function graphs $y=f(x)$ provide a particular example, but we can also handle more complicated curves here, for example the unit circle ($F(x,y)=x^2+y^2-1$).

Let us draw a part of the curve that goes under the name the Trisectrix of Maclaurin, a curve that can be used to trisect angles (named after Colin Maclaurin in 1742). We use the function

$$F(x,y)=2x(x^2+y^2)-(3x^2-y^2).$$

Let us construct the path and draw it.

```
\startluacode
    local N   = 1000
    local xx  = 3/N
    local yy  = 3/N

    local function f(x,y)
        local x = xx*x - 1
        local y = yy*y - 1.5
        local z = 2*x*(x^2 + y^2) - (3*x^2 - y^2)
        if z > 0 then
            return '1'
        else
            return '0'
        end
    end

    potrace.setbitmap("mybitmap", potrace.contourplot(N,N,f))
\stopluacode

\startMPcode
    path p ; p := lmt_potraced [
        stringname = "mybitmap",
        value      = "1",
        tolerance  = 0.1,
        threshold  = 1,
        optimize   = true,
    ] ;
    p := p xsized TextWidth ;
    draw p withcolor "darkred" ;
    drawpoints p withcolor "orange" ;
    drawpointlabels p ;
    currentpicture := currentpicture ysized .4TextHeight ;
\stopMPcode
```

The leftmost graphic in figure 29.4 shows the result of this code. We also draw the points and the point labels. This way we can easily find out which part of the path to draw. In this case it seems that we need the first 34 points.

```
\startMPcode
    path p ; p := lmt_potraced [
        stringname = "mybitmap",
        value      = "1",
        tolerance  = 0.1,
        threshold  = 1,
        optimize   = true,
    ] ;
    p := subpath(0,33) of p ;
    p := p xsized 4cm ;
    draw p
```

```
        withcolor "darkred"
        withpen pencircle scaled .5mm ;
    currentpicture := currentpicture ysized .4TextHeight ;
\stopMPcode
```

You need to keep the resolution (determined by the input) and therefore scaling in mind and choose the pen accordingly as in the second drawing in figure 29.4.

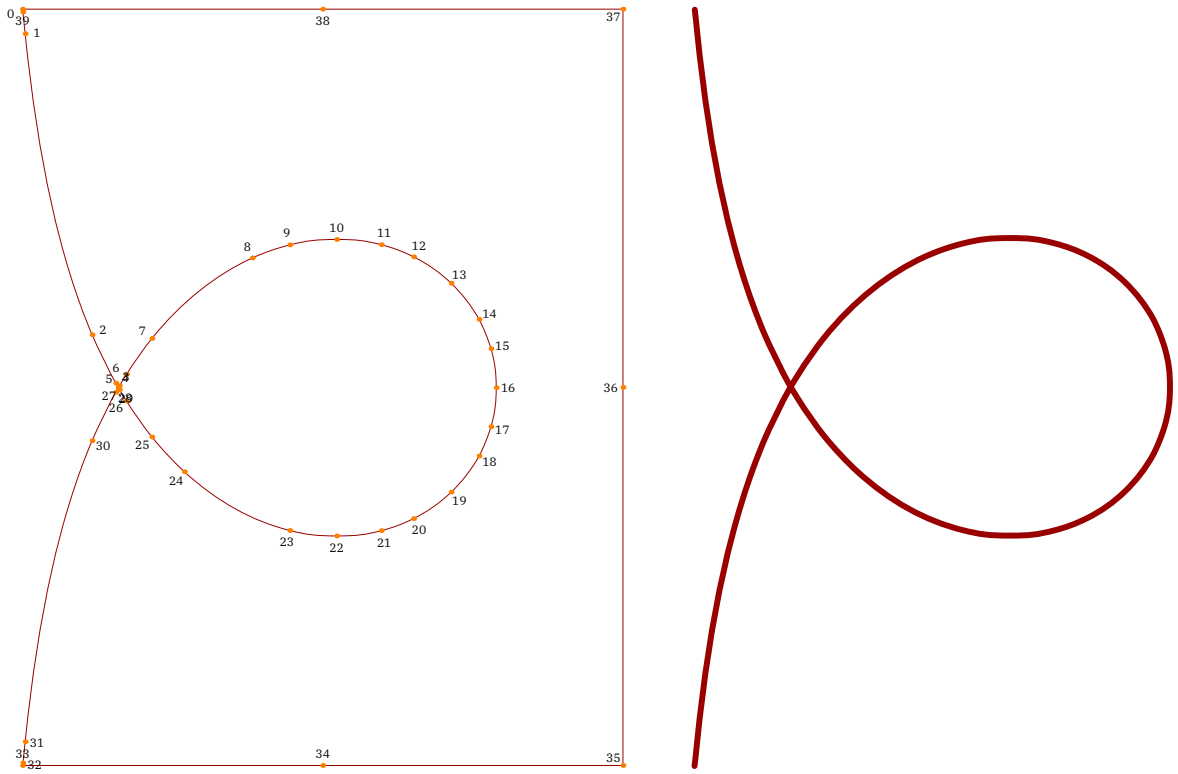


Figure 29.4 The same contour plot with different details.

This method of drawing contour plots is efficient, and it gives, in contrast to some traditional methods, curves that look smooth, with relatively few points. It also has weaknesses, one of them being that the inequality $F(x, y) > 0$ does not always single out the curve $F(x, y) = 0$; it might be the case that the function F is positive on both sides of the curve $F(x, y) = 0$.

We invite the reader to be creative and play with this (to us) new toy. Have fun!

29.8 Coda

And, in the spirit of having fun, here are some final images. This first one is a photo of a 'tool' that we came up with at the ConTEXt meeting. Willi Egger made a kit for the attendees, so there was the usual cutting and glueing involved. This kit is shown in figure 29.5. The dots are seeds. It also shows an emoji that Mikael's children made when we were playing with this feature. The first image has the rendering of that input, while the second 'explodes' the pixels in the x direction (so columns are duplicated), resulting in a more symmetric image.

We only show the second emoji. When we set `ny` too, we get a still interesting but somewhat weird face instead. You can try that yourself.

```
draw image (
    lmt_startpotraced [
```

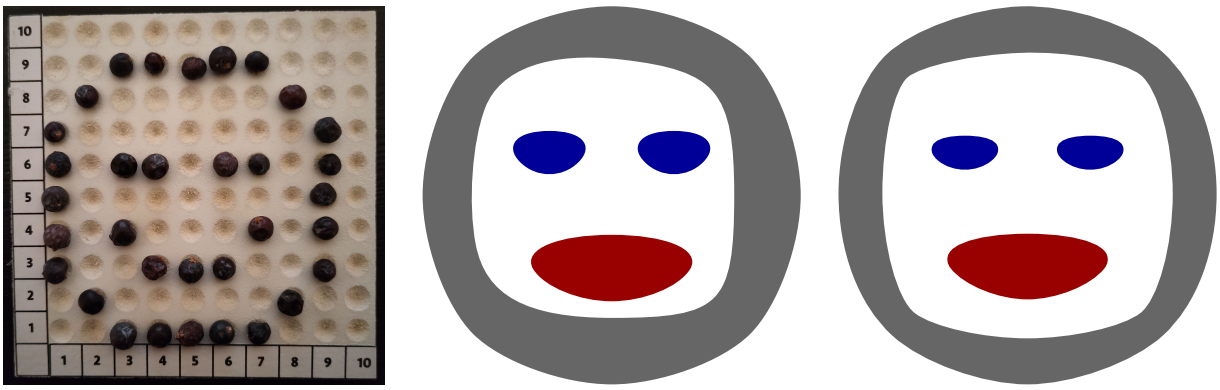


Figure 29.5 A bitmap kit by Willi Egger.

```
        nx    = 2, % more symmetric, 3 gives different curves
     %  ny    = 3, % clown
        bytes =
            "..11111..
             .1.....1.
             1.......1
             1.22.22.1
             1.......1
             1.3...3.1
             1..333..1
             .1.....1.
             ..11111.."
    ] ;
    fill lmt_potraced
        [ value = "1", size = 1 ]
        withcolor "darkgray"
        withpen pencircle scaled 0.1 ;
    fill lmt_potraced
        [ value = "2", size = 1 ]
        withcolor "darkblue"
        withpen pencircle scaled 0.1 ;
    fill lmt_potraced
        [ value = "3", size = 1 ]
        withcolor "darkred"
        withpen pencircle scaled 0.1 ;
    lmt_stoppotraced ;
) sized 5cm ;
```

30 Crossing rivers

Occasionally you can hear or read a TEXie argue that the engine should be able to deal with rivers. Now, to be honest, we never really bothered much about that, if only because one should rewrite the content if one is really disturbed, but we come to that. The excursion below is a side track of introducing some new graphic capabilities in ConTEXt while at the same time looking at possible improvements in the paragraph builder.

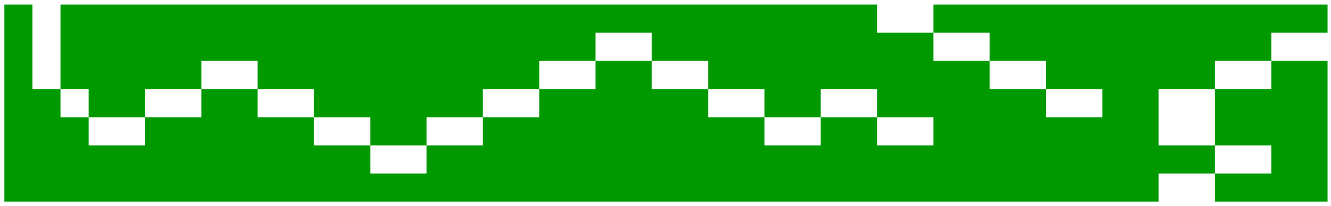
Let's start with some terminology. In `tex.web` you can find Don Knuth saying: "After considering TEX's eyes and stomach, we come now to the bowels" and I've heard talks at TEX meetings where the presenter impresses the audience by using these terms. The same happens with talking about widows, clubs, penalties, demerits and such. Talking rivers fits into this phenomena. The fact that after decades of TEX usage no one really bothered much about it beyond mentioning and suggesting half-way solutions is a telling sign.

If one talks about a river in a typeset text what is actually meant is that there are spaces in successive lines that when you look at it seem to be connected and that is considered bad or at least annoying. Of course there is nothing that prevents us to consider (say) `m`'s or `oo`'s stacked in successive lines to be seen as mountain rig but let's stick to spaces.

You can wonder what is so bad about a gently meandering river. In fact, the more it meanders, the less it will be noticed. It's canals that matter more. In a paragraph with wide spaces due to the constraints one can be annoyed by those, and we will call them ponds. When they become too large we can talk about lakes. Actually, when rivers are narrow, it would be better to talk of creeks.

So, in order to please those who love this analogy speak, we now have a decent repertoire of seemingly bad things to look at: ponds and lakes, creeks and rivers, and canals. An empty page is basically an ocean and too much white space between paragraphs might be seen as a sea, but that can be intentional.

So how bad is all this in practice? Before we show some examples, we make a decision: a pond is as bad as a creek, and a lake as bad as a river. Canals are even worse but they will often show up like rivers: it is mostly a visual thing. A river can meander which means that it can backtrack: a real worse case is a paragraph where it meanders over multiple lines horizontally.



Anyway, we distinguish three different cases small (ponds and creeks), large (lakes and rivers) and whatever sits in between. In the next example we see these three cases, where blue is small and red is large.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

The question now is, are these red rivers really that bad? Let's see how the text looks when we don't identify them:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We bet no one is really bothered by this. In fact, one can be more annoyed by the commas at the end of nearly every line. This text (a quote from Tufte actually) is also a bit atypical because of the many words separated by commas but that makes it such an excellent quotation for many tests. There's always something a user can complain about.

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

In the previous example we are in a way worse situation: plenty of creeks and rivers there, but again, you won't hear us complain. So can we make an example that will make us feel bad? In figure 30.1 we see something that will never (or at least seldom) happen in a TEX document: we see no hyphenation.

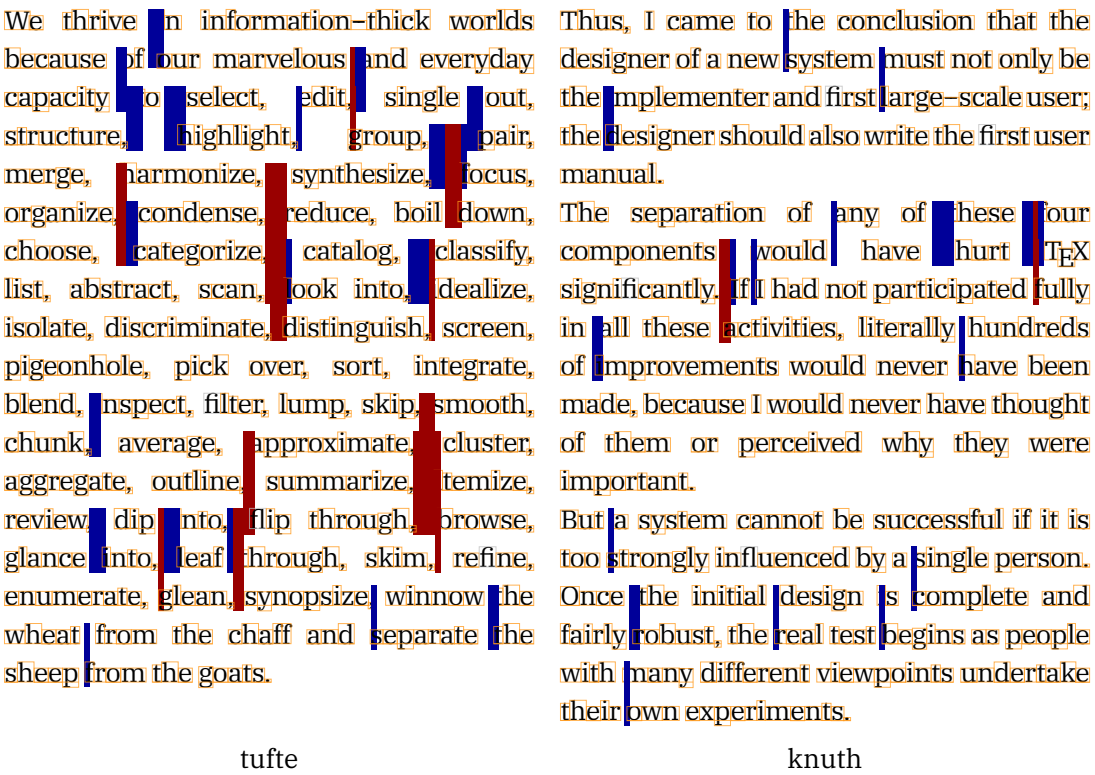


Figure 30.1 Overflooding

By now you might wonder how we get to these areas. What happens here is that we first turn the typeset result into a (sort of) bitmap. Every line gets the same number of slots, each representing 2pt advance. We start out with all slots being zero but then assign slots that overlap with interword spacing to one. We can then run over the rows and see where what columns have value one. But that is not what we do. Because we want a picture, and because we also want meandering visualized, we feed the bitmap into potrace, a library that is built in LuaMetaTEX, and convert the resulting vector representation into a MetaPost graphic.

If one is only interested if there are rivers at all, one can set the criteria high and look only at the red blobs. In fact, if there is more than one red blob one can already decide that we have a bad situation. But what can we do about it? Well, not that much. Without hyphenation one is kind of lost, with hyphenation one can best rewrite the paragraph and hope for the best. Of course a way out is to use a new feature of LuaMetaTEX (and ConTEXt): additional emergency par builder steps. These come cheap and just try to do a better job by changing constraints, one of which is selective expansion (aka hz). In fact, trying to get rid of rivers and lakes by tweaking can be considered as harmful as messing with looseness, a trick that one only applies when the results can be visually checked. As with real rivers, messing with them will just make things worse. Let's consider them natural boundaries that we don't even bother to cross.

```
01000000000000000000000000000011000000000000000
01000000000000000000011000000000011000000000011
01000001100000000001100110000000001100000001100
00100110011000000110000001100110000001100110000
00011000000110011000000000011001100000000110000
00000000000011000000000000000000000000000001100
00000000000000000000000000000000000000000110000
```

Just in case you're curious about what bitmap gets fed into potrace, we show it above. You might see rivers, or not. And it might be clear now that we are not going to waste time on this in the perspective of improving the par builder. It would be one of these tuning options that in the end no one would use, just because "fixing this" might as well "break that" which means that such an automatism will demand closer inspection of the output than a user wants. You might also realize that we don't really need potrace for detecting lakes and rivers but it just gives nice graphics, and that's what we're after. One can even argue that if avoiding any river is the objective, even one pair of vertical ones is already enough to not look further and declare the whole paragraph to be bad which then has to trigger a new attempt. It is however not that hard to imagine something `\riverdemerits` and maybe even `\doubleriverdemerits` based on a rather dumb fast pairwise check. The complication of adding this to the par builder is that we then need to package all the possible lines and that's not going to happen any time soon.

What might get unnoticed here is that we not only look at inter-word spacing but also treat small glyphs like punctuation followed by such spacing as part of a river and we can imagine plenty more cases to consider. It just shows that if spacing is a problem, anything else can become one. We can also add some margin to a space because being a bit more or less tolerant can matter.

Just in case you want to experiment with this, here how it goes:

```
\startshowrivers[criterium=6]
    ...
\stopshowrivers
```

where the following optional parameters can be set:

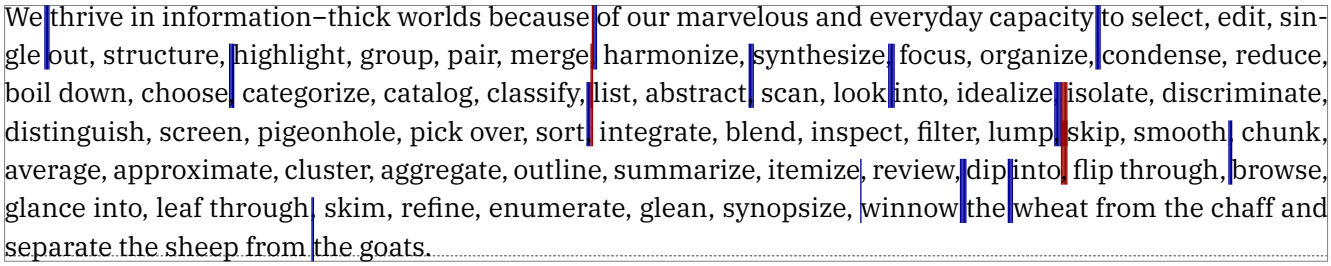
```
step            2pt
margin          0pt
height          .2\bodyfontsize
depth           .2\bodyfontsize
spaceinbetween  .1\lineheight
criterium       4
```

Here the `step` determines the accuracy, the `margin` is what gets added to the space skip when it is encountered, the `height` and `depth` are used to identify for instance punctuation characters that are treated like

glue. When we have a vertical glue that exceeds `spaceinbetween` we assume a new paragraph is started. Finally, `criterium` is what makes the red shapes show up; steps 1 and 3 result in the green and blue ones.

Of course this is mostly an attempt to show that in practice we shouldn't care too much about these rivers. As with quite some nice features, in the end once they are there one forgets about them because in retrospect they were not that much needed. This is why we only mention that there is a simple and fast variant that one can enable with `\showrivers` which acts upon the page but it is less fun to look at.

Let us end by mentioning that we downloaded the Henry James book "The turn of the screw" from the Project Gutenberg web page and compiled it in many different page layouts and with different choices of fonts, and it was not so easy to produce any disturbing rivers. TEX does a very good job. The bigger space after the period could be a bit disturbing, in particular if it appeared on consecutive lines and if the lines had different stretch.


We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

31 Including PDF files

Rendering glyphs in a pdf happens based on information in the page stream. In such a stream we find font triggers that use identifiers like F1 and afterwards placement operators inject shapes by referring to an index in the font, say hexadecimal 0003, and that index can refer to any shape. The identifier is resolved via the Font entry in the Resources dictionary of the page:

```
5 0 obj
<<
    /Contents 3 0 R
    /Resources << /Font << /F1 1 0 R >> /ProcSet 2 0 R >>
    ...
    /Type /Page
>>
endobj
```

Following the F1 reference we end up at:

```
1 0 obj
<<
    /BaseFont /TNTUFI+LMRoman10-Regular
    /DescendantFonts [ 11 0 R ]
    /Encoding /Identity-H
    /Subtype /Type0
    /ToUnicode 14 0 R
    /Type /Font
>>
endobj
```

and when we then descend into the first entry in the DescendantFonts array we come to:

```
11 0 obj
<<
    /BaseFont /TNTUFI+LMRoman10-Regular
    /CIDSystemInfo << /Ordering (Identity) /Registry (Adobe) /Supplement 0 >>
    /FontDescriptor 9 0 R
    /LMTXRegistry 8 0 R
    /Subtype /CIDFontType0
    /Type /Font
    /W 10 0 R
>>
endobj
```

The W entry value, rather short relative to the other keys, refers to an array object that holds the widths of the glyphs so that the viewer knows how much to advance; the FontDescriptor points to the shapes; and LMTXRegistry will be discussed later. These examples come from typesetting:

```
\starttext
    \startTEXpage
        test
```

```
    \stopTEXpage
\stoptext
```

With LuaTEX we see this in the page stream:

```
BT
/F1 11.955168 Tf
1 0 0 1 0 0.11949 Tm [<0069003200620069>]TJ
ET
```

and also this in the mapping from index to Unicode, which is object 14, defined by the `ToUnicode` value shown above (I added the characters as comments):

```
3 beginbfchar
<0032> <0065> % e
<0062> <0073> % s
<0069> <0074> % t
endbfchar
```

When we use LuaMetaTEX instead we get:

```
BT
/F1 10 Tf
1.195517 0 0 1.195517 0 0.065717 Tm [<0001000200030001>] TJ
ET
```

and:

```
3 beginbfchar
<0001> <0074> % t
<0002> <0065> % e
<0003> <0073> % s
endbfchar
```

Thus, where LuaTEX uses the original index in the font (not to be confused with the character's Unicode value, if it has one at all), in LuaMetaTEX, or more accurately with the ConTEXt backend, we start at one and number upwards. This gives smaller files.

The only way to find out what an index is actually referring to is to consult the abovementioned `ToUnicode` vector in the font resource because there we map from index to Unicode. That information is used when you search in a pdf file or cut-and-paste from it. Because glyphs can be unrelated to Unicode, and because multiple glyphs can share the same Unicode slot, the index is what makes a glyph unique.

When a (page from) a pdf file is included in a document LuaTEX will copy the relevant objects to the main file. Of course the page itself is copied (with the page stream making up the content). Copying is also driven by the `Resources` key in the page dictionary. In addition to the `Font` list we've seen above, there can also be an `XObject` array and its entries need to be copied as well. This copying is recursive because the resources themselves can point to objects with resources. It is quite normal in a pdf file to share resources. The rendered glyphs, for instance, come from fonts that contain the shape definitions and basically these are the same, independent of scaling.

The index can be the original glyph index in the font but, because we subset, it can also be a different one, depending on what gets included. This is illustrated in the example above. By default the LuaTEX engine

just copies and doesn't worry about what gets copied. However, in MkIV we can load some code that plugs into the LuaTEX backend and is thereby capable of merging fonts from the included pdf (page) with one used in the document. This process is driven by setting the `compact` key in `\externalfigure` to the value `yes`. Here we assume that the references to glyphs in the page stream are the original (or equivalent) indices but this is not guaranteed to be true. We can check a little by comparing the Unicode mapping as well as doing a visual check afterwards, but neither are robust. It still works ok as long we use exactly the same font; essentially, we check the name and when it matches we force the glyph into the current file and use the font reference of main document for the embedded reference instead.

In LuaMetaTEX we do it differently and there are reasons for this. First of all, we have different numbering in the main file and inserted file, so we cannot use the indices directly. In addition, we have to also take care of Type3 fonts that refer to fonts that we merge (we use these fonts in, for instance, math delimiters). Finally we cannot simply look at the name because we can have a variable font instance that has different axis properties. So we have to be more clever: we need to parse the content streams of the page, XObjects and charprocs to find out what glyphs are referenced and replace indices when applicable. In addition we consult some extra information that is included when ConTEXt did typeset the (to be embedded) file. That information contains a stream index to original index mapping, and also has a variable font recipe if needed. There is also some additional information so that we at least check if we have the same font.

			native	`compact=no`	`compact=yes`
LuaTEX	**MkIV**	**compressed**	839 KB / .20 sec	731 KB / 0.20 sec	231 KB / 0.22 sec
		decompressed	1784 KB / .20 sec	943 KB / 0.22 sec	426 KB / 0.23 sec
LuaMetaTEX	**MkXL**	**compressed**		544 KB / 0.18 sec	147 KB / 0.24 sec
		decompressed		783 KB / 0.16 sec	552 KB / 0.19 sec

we compare three alternatives. The native inclusion in LuaTEX leaves the work to the engine. When the plugin is loaded, we will use the Lua-based inclusion code which is a bit more clever in sharing objects. In the LuaMetaTEX variant we also copy objects into the main file but there we have no plugin and sharing happens anyway. Here with `compact=yes` we also merge fonts but this time based on parsing the streams. This parsing is more demanding and bumps runtime but is also more rewarding in terms of file size. For completeness we show the results with and without pdf object stream (zip) compression. The need to decompress and compress also has some impact on performance.

In case the slightly slower inclusion in LuaMetaTEX bothers you, there might be some comfort in knowing that the 20 files accumulate to 564 KB and a fresh run takes 49 seconds. When we use LuaTEX the file size total bumps to 748 KB and the initial runtime goes up to 94 seconds. So in the end the LuaMetaTEX-based variant is more efficient.

So, in MkIV there are two reasons for having the plugin. The first is that by sharing common objects we can, for instance, include many pages from the same document with little overhead. For this, compact doesn't need to be active. However when for instance we include many documents we can see that merging fonts does pay off handsomely. In MkXL we already have the first benefit (sharing) by default and here we can also do better by merging fonts. Because that merging is more aggressive you see better numbers in the table for MkXL.

A practical usage scenario is making manuals where we process examples (using ConTEXt buffers) in independent runs so that they are independent from the main document. Of course there is only a gain if these examples share fonts with the main document or with each other. Here is the test case:

```
\startbuffer[common]
    \usebodyfont  [dejavu]
```

```
    \usebodyfont  [lucida]
    \usebodyfont  [bonum]
    \setupbodyfont[modern]
    \setupalign[tolerant,stretch]
\stopbuffer
```

We load four different font sets in a common buffer but also use them in the main document:

```
\getbuffer[common]
```

We define two additional buffers that each create a document with two pages. We use different languages because otherwise there is little to merge, as the sample texts use the regular Latin script:

```
\startbuffer[demo-1]
    \start
        \switchtobodyfont[dejavu]\samplefile{ward}
    \stop \blank
    \start
        \switchtobodyfont[lucida]\samplefile{davis}
    \stop \page
    \start
        \switchtobodyfont[bonum] \samplefile{knuth}
    \stop \blank
    \start
        \switchtobodyfont[modern]\samplefile{tufte}
    \stop \blank
\stopbuffer

\startbuffer[demo-2]
    \start
        \switchtobodyfont[dejavu]\mainlanguage[es]
        \samplefile{cervantes-es.tex}
    \stop \blank
    \start
        \switchtobodyfont[lucida]\mainlanguage[sv]
        \samplefile{alfredsson-sv.tex}
    \stop \page
    \start
        \switchtobodyfont[bonum] \mainlanguage[de]
        \samplefile{aesop-de.tex}
    \stop \blank
    \start
        \switchtobodyfont[modern]\mainlanguage[cz]
        \samplefile{komensky-cz}
    \stop \blank
\stopbuffer
```

Optionally we enable compact inclusion:

```
% \setupexternalfigures[compact=yes]
```

Here is the main document:

```
\starttext
    \dorecurse{10}{
        \startTEXpage[offset=1ex]
            \start \switchtobodyfont[dejavu]\samplefile{ward}  \stop \blank
            \start \switchtobodyfont[lucida]\samplefile{davis} \stop \blank
            \start \switchtobodyfont[bonum] \samplefile{knuth} \stop \blank
            \start \switchtobodyfont[modern]\samplefile{tufte} \stop \blank
            \setbuffer[#1]#1\endbuffer % #1 is the iterator
            \hbox\bgroup
                \typesetbuffer[common,demo-1,#1][width=10cm,page=1]
                \typesetbuffer[common,demo-1,#1][width=10cm,page=2]
            \egroup
            \blank
            \hbox\bgroup % these are processed in sperate runs
                \typesetbuffer[common,demo-2,#1][width=10cm,page=1]
                \typesetbuffer[common,demo-2,#1][width=10cm,page=2]
            \egroup
        \stopTEXpage
    }
\stoptext
```

In order to get ten times two unique documents to be included, we smuggle an extra buffer into the subsidiary runs. We need to do this because otherwise the hashes of the content of these sub-documents are the same and we'd end up with only two documents and successive inclusions would share these. Of course the first run with fresh buffers will take more runtime because the sub-documents need to be processed (twice in order to get multi-pass activities resolved).

There are a few pitfalls. First of all we have to make sure that we only merge references to the same font. This is often no problem as long as we don't update fonts with different shapes in the same slots but we can assume that the version number is different then. For the application we have in mind, buffered sub-runs or inclusion in related documents that get processed in a short time span, we are probably ok. We can make the check more tolerant or more clever, and might do that in the future. On the average the inclusion is already rather efficient when a few pages from a few documents are used.

A second pitfall is that when we improve the ConTEXt (font) backend we can have better shapes or more precise metrics but because metrics are unlikely to change much in the glyph programs we're probably okay. Even mixing the more efficient so-called compact font mode with normal font mode (not to be confused with compact inclusion) should work out well enough. In case of doubt: trust your eyes or just regenerate the documents involved in the inclusion.

Finally it is worth mentioning that there is a noticeable overhead but if becomes necessary I can optimize the handling of the stream a bit by replacing the more general stream parser by a dedicated one for this purpose.

Let's stress one thing again. Because shared font usage will never be (guaranteed) watertight you do need to check visually. A bad merge will immediately show up by the included image rendering with garbled text. There are some extra safeguards in the MkXL approach that are absent in the MkIV solution which is why the latter is considered an experiment and not loaded by default. I could spend time on it but as we moved on to LMTX (the MkXL LuaMetaTEX combination) it makes little sense.

Does the story end here? Not entirely. Occasional validation requirements have a side effect that some users have to fix old pdf files to suit demands. Let's mention a few issues users run into:

- Embedded so-called Type0 fonts can lack a `/CIDSet` and/or `CIDToGIDMap` entry that needs to be added.
- A document can refer to external files that are not embedded. Normally these are in `WinAnsi` encoding and page stream indices match encoding indices so we can smuggle these files into the document.
- Font resources can be embedded multiple times with different subsets, likely per page. Different names are used, which complicates matters, but it makes sense to try to merge them.
- Fonts with the same name but in Type1 as well as TrueType format, both using the original indices, occur in the same document so they can be merged.

We can deal with this quite well if we have the original fonts available. Failures to do this well immediately show up so we can again trust our eyes. In an automatic large scale fix operation we can built in some safeguards.

Then, as we're fixing included pages anyway, we may as well try to conform to standard even better, for instance:

- Instead of gray scales cmyk and rgb colors are used, or one of these color spaces is not handled right and we need to remap colors. It can be a side effect of lazy programming in producer software.
- Extended graphic states are used, for instance for transparencies while there is no transparency actually being used. These can be side effects of producers just emitting as much as they can.
- Irrelevant grouping can be applied to pages and `/XObjects`. In fact, when a validator complains we can just as well get rid of them.

For such things we need to fix the `/Resources` as well as the page content streams so it adds a little more overhead but when we just convert documents it is a one-time effort so a few more milliseconds won't be a big burden.

But discussing these details doesn't really fit in this font discussion so we end by mentioning that we have a framework in place for plugging in fixers of any kind. The approach is to configure what standard to use and what fixes to apply. Only time will tell if this is sufficient. Most users probably never have to worry about it anyway.

Many thanks to Karl Berry for copy-editing this text!

32 Signing documents

How did we end up here

Here I discuss a feature of pdf documents that TEX users can safely ignore most of the time but that is part of the package. But before we arrive at describing what it is, first a few words about pdf and the way it is used.

When pdf showed up as a format, dvi was what TEX users had to deal with. It was possible to preview the result with a dvi viewer or convert it to some format suitable for a printer. The dvi format is rather minimal and has no resources like fonts and images embedded. Basically the file only positions glyphs and rules on a canvas and the glyphs are references to external fonts. Color and similar effects have to be implemented using the `\special` primitive that puts directives for the backend driver in the file. Although technically one could embed fonts and images using specials and thereby create a self-contained file it never happened, partly because it demands a dedicated viewer and (definitely at that time) increased runtime.

At that time PostScript was one of the popular output formats. For a long time I used dvipsone for (robust) printer output (with outline fonts) and dviwindo for previewing (because it was fast, supported color, graphics and hyperlinks). The initial road to pdf was to add another step to the conversion: using Acrobat to convert a PostScript (enhanced with so-called pdfmarks) into a pdf file. Later direct dvi to pdf converters showed up alongside pdfTEX that integrated an alternative backend. We can realize that without these more direct methods TEX would not be as popular as it is now. The fact that in ConTEXt we had a rather generic (abstract) backend, where specific drivers plugged in, indicates that we didn't foresee that pdf would eventually take over.

One of the reasons pdf showed up alongside PostScript is that it removes the interpretation part from the end result. Where PostScript is a programming language and the result needs to be interpreted in the printer or in a viewer (like Ghostview), pdf is a collection of related objects that express what gets rendered in what way. Often the pdf files are smaller, also due to compression (so they transfer faster), and the lack of additional processing removes a bottleneck in high speed and high resolution printing. If you look at it this way, pdf is primarily a *printer format*.

Some tools in the Adobe suite of editing programs use(d) a mix of binary and PostScript to store the state. At some point pdf became the container format, although one could find curious mixes of PostScript, pdf, even configurations stored as typeset streams, but that might have been normalized by now. So, from this perspective pdf is a *storage format*, kind of an object-oriented one.

When pdf showed up it had some rudimentary support for annotations, like hyperlinks, and also for embedding of audio and video. I'm pretty sure that TEX was among the first programs to support this. Over time more annotation types showed up, like widgets (forms), complex media and 3D, JavaScript, comments and attachments. Widgets evolved a bit and one has to play rather safe (and not use all features) because some bugs and side effects became features and there is limited support in viewers, if only because often JavaScript is assumed. Media have always been sort of a mess, especially when the straightforward annotations were dropped. We have always supported them but I consider them unreliable in the long run. Most hyperlinks are working as expected, comments (a form of pdf annotations) when used wisely also work ok, although that depends on the viewer (interfaces change). No matter what one thinks of all this, here pdf is definitely a *viewing format*. Because comments can be added, pdf files can also be used in redacting workflows.

It is also worth remembering that in the early days only Acrobat was available for viewing; there was even a version for msdos. The Reader only offered basic functionality and for the real deal one needed Exchange,

which didn't come free. There was a complicated scheme for shipping a special version with documents: basically that was the business model for using pdf for an on-screen reading experience. This was not a success (cumbersome as well as expensive), and when Internet access became more common, using cdrom for distributing documents was soon obsolete, let alone installing a related closed source and operating system dependent viewer. We'll see that with document signing, another attempt at a business model is made.

Following up on that we now arrive at two additional aspects. One is accessibility and I could spend a whole article on that. Let's stick to the observations that the idea is that rendered content can be reinterpreted for reading aloud, for reflow (sic), maybe for cut-and-paste. Personally I wonder why one would use pdf to provide adaptive accessibility, because html is meant for that. Distributing a structured source might be more efficient when interpretation is needed. Anyway, it's not that hard to support but we end up with a bloated pdf file, while the pdf format started out with attempts to minimize size. I understand that publishers don't like to distribute sources but if this is the solution one can wonder. The second addition is to render and present text in a way that cannot be tampered with and this is where signing comes in: can we somehow mark a document such that the receiver can trust its content. More about that later, but what we can say here is that pdf has become a *distribution format*. Conforming to standards, tagging, encryption and signing all play a role in this.

No matter what usage we consider, all of them depend on reliability. Can we show and process a document in the future? Can pdf be relied upon as a long-term archival format? From that perspective, standardization has to be mentioned. Already early in the history of pdf, plugins (and additional workflows) provided the printing industry ways to check if a file was okay. One should think of color spaces being used, fonts being present, etc. Some tools manipulated the pdf, not always with the best outcome, but we leave that aside. Other tools were a bit more tolerant than might be considered healthy, for instance by ignoring a bad xref table (basically the registry of objects in a pdf file) and either fixing it or just generating one from scratch. Although Acrobat can complain or fail to open a document, on the average commercial and open source tools are tolerant enough and the lack of a proper error log means that the pdf generators don't get fixed when that still can be done. It is also good to keep in mind that there is a whole industry around pdf generation, validation, manipulation, etc. and huge money making machines are not always on the retina of, for instance, TEX users who produce pdf. Of course by now there are so many documents in pdf format around that being tolerant kind of comes with the package. Validators like VeraPDF evolve and a document that is ok today (2023) might fail the test tomorrow, and the verdict even depends on the pdf framework being used (there are options). Where TEX users can often regenerate a document from source this is not true for the majority of documents produced elsewhere.

It is also important to notice that rather soon in the history of pdf, Ghostscript became an option for viewing and at some point commercial and open source viewers showed up. Not all were perfect and even today there are differences in quality and functionality. A good test is how well cut-and-paste deals with spaces and how well a test area gets selected. The open source viewers are slow in catching up, but because the evolution of media pdf annotations isn't that stable either for most purposes viewers like SumatraPDF (MS Windows) or Okular (linux) is what I use today, especially now that Acrobat has moved to the cloud. There is also some competition from browsers that show pdf. For purposes of signing (which we'll get to next) one probably has to rely on Acrobat for a while, but we'll see.

So, what does signing bring to this? Digital signatures have been around for a while. You can for instance sign a document with a certificate (similar to what secure webservers do with sites). In that case the distributed blob has security-related information as well as the content. A validating application can take the content and check if it has been tampered with. It can do so off line (with limited security) but also go online and check the embedded certificate. With signed pdf files the same is true apart from the fact that here the signature is a partial one, not embedding the data. Instead the signature is embedded in the pdf file.

Before we move on we have to stress that signing is not the same as (password protected) encryption. A signed pdf file is by default just readable, unless one explicitly encrypts the file. These processes are independent. Here we ignore encryption; suffice it to say that ConTEXt can do it, but apart from users asking for it I don't know if it ever gets applied. We discuss the process of signing in the perspective of ConTEXt, although in itself it is not bound to that macro package.

Let's first look at how text ends up in a pdf file. Take this source file, in ConTEXt-speak:

```
\startTEXpage[offset=1dk]
   some text
\stopTEXpage
```

This leads to a so-called page stream that contains this (except normally you are likely to see garbage because compression is applied, so decompress first):

```
BT
/F1 10 Tf
1.195517 0 0 1.195517 7.485099 7.616534 Tm
[<0001000200030004000500060004000700060>] TJ
ET
```

We switch to a font (`Tf`) with id `F1`, set up a text transform matrix (`Tm`) and render the four plus four characters (`TJ`) indicated by their index into a (in our case subsetted) font. The `0005` is not really a character: it refers to a space. It looks unreadable but one can figure out the text by consulting the `ToUnicode` resource associated with the font as it has the mapping from the index numbers to Unicode (with comments added):

```
<0001> <0073> % s
<0002> <006F> % o
<0003> <006D> % m
<0004> <0065> % e
<0005> <0020> %
<0006> <0074> % t
<0007> <0078> % x
```

There is nothing hidden here and one can actually even change the text by changing an index in the page stream, although of course you can only use indices that are available and you also have to accept weird rendering due to the change in progression when the referenced glyph has different dimensions. More extensive tampering with the document has more severe consequences. For instance, the page object looks like this:

```
3 0 obj
<< /Length 118 >>
stream
...
endstream
endobj
```

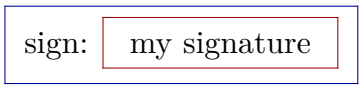
So changing the content also demands changing the `Length`. Even worse, there is an entry in the object cross reference table:

```
0000000075 00000 n
0000000244 00000 n
```

that needs to be adapted, including all following entries. So, tampering is possible but not something that is likely to happen. Nevertheless we continue as if some guard against this is needed. We now assume the following document:

```
\setupinteraction[state=start]

\definefield[signature][signed]

\defineoverlay[signature][my signature]

\starttext
    \startTEXpage[offset=1ts,frame=on,framecolor=darkblue]
        sign: \inframed
          [background=signature,framecolor=darkred]
          {\fieldbody[signature][width=3cm,option=hidden]}
    \stopTEXpage
\stoptext
```

We get a small, one page, document:



This document has a widget that looks like this (some less relevant entries are omitted):

```
2 0 obj
<<
    /Type    /Annot
    /Subtype /Widget
    /FT      /Sig
    /T       <feff007300690067006e00610074007500720065>
    /V       1 0 R
    /Rect    [ 38.445125 11.522571 123.484489 23.471172 ]
>>
endobj
```

In principle we could add an appearance stream and decorate the widget but when adding signature support to ConTEXt I found that using a parent-kid approach, for instance, was not appreciated by some programs (I used mutool (mupdf), pdfsig (poppler), Okular and Acrobat Reader for some basic testing), so in the end the `V` key ended up in the root widget. It probably relates to fuzzy specifications, experiments with specific tool chains, non-public validation processes, etc. Round trip signing and verification seems not entirely trivial, so best to play safe.

When the value of a `Sig` widget is a string, signing is up to the viewer but when we have a dictionary the signature can be in the file. The `V` value of `1 0 R` is a reference to a dictionary with object number `1`. Here is what that value looks like when we generate this document:

```
1 0 obj
<<
    /ByteRange [ 2000000000 2000000000 2000000000 2000000000 ]
    /Contents  <00000000000000000000....00000000000000000000>
    /Filter    /Adobe.PPKLite
```

```
    /SubFilter /adbe.pkcs7.detached
    /Type /Sig
    >>
endobj
```

The `Filter` and `SubFilter` entries are sort of default, though alternatives are possible, which then requires additional information to be added and also a viewer (or validator) able to deal with it. We leave that aside. The `Contents` hex-encoded string is a placeholder for the signature and in our case is 4096 bytes long. We could compute a bogus signature and check the size instead. Here the `ByteRange` and `Contents` are actually invalid but viewers are (supposed to be) tolerant so it triggers no error. After all this widget is only consulted when signatures are checked. The general structure of the file is like this:

```
%PDF-1.7
....
1 0 obj
<< /ByteRange [ .... ] /Contents <....> .... >>
endobj
....
xref
```

The signature ends up between `<` and `>` and has to be calculated over the bytes specified by the `ByteRange` entries. Although one might think that these can be arbitrary, in practice it looks like it is best sticking to the recommendation:

```
start of file
length upto position < of contents
position after > of contents
length upto end of file
```

So basically all except the `Contents` value is taken into account. Because the ranges are part of that, they need to be filled in properly, something that has to be done after the pdf file is finished because only then is the size known. In ConTEXt that could be done as part of the main run, but it makes little sense because we need to adapt the file anyway. If the file is called `sign-001.tex`, we get this:

```
mtxrun --script pdf --sign --certificate=sign-001.pem --password=test sign-001
```

The script will set the byte ranges and fill in the content. It does that by making a data file and running `openssl` with the appropriate parameters, although with `--library` one can avoid the temporary file and gain a bit. Just for the record: we don't depend on that library but have only a minimal delayed binding to a few functions, with Lua wrappers so it has no impact on (compiling) the LuaMetaTEX binary. Eventually one ends up with something like this (values abridged):

```
1 0 obj
<<
    /ByteRange [ 0000000000 0000006276 0000010375 0000000380 ]
    /Contents  <3082061a06092a864886....010702a082060b308206>
    ....
endobj
```

Verifying can be done as follows:

```
mtxrun --script pdf --verify --certificate=sign-001.pem --password=test sign-001
```

which reports:

```
sign pdf | signature in file 'sign-001.pdf' matches the content
```

while changing a byte in the trailer id results in:

```
sign pdf | signature in file 'sign-001.pdf' doesn't match the content
```

For verifying we can load the pdf file and use the `ByteRange` specification but for signing this is less trivial: when we load a pdf file we load a structure that is ignorant of the position in the file. We could use the cross reference table to find the position in the file of the object but that assumes that this table is available. So here we have two alternatives. We can write an auxiliary file (`sign-001.sig`) at the end of the TEX run that has the relevant information. This approach permits us to keep the pdf file simple: we reserve enough characters for the ranges and content so we can overwrite them. If the file is lacking, the sign routine tries to locate the object in the pdf from the list of widgets and once we know its number we also know where the object is in the file. This alternative adds a little overhead because at least the cross reference table has to be loaded. Whatever route we take, it is still prettier than appending additional objects to the file and basically creating a new version, which not only makes the file larger but also keeps unused objects around. Applications like mutool and Acrobat prefer that route, though, in part because they add their own appearance streams.

We now need to discuss these certificates and that is where it becomes less convenient. For testing, I use a Let's Encrypt certificate but these officially cannot be used as they are flagged as web certificates. There is (what's new here) a whole industry behind this signing. You need to get a certificate someplace and for that often have to sign up for a yearly subscription. In the worst case you get a token instead of a file and then have to set up some delegated workflow. Feeding a document into a usb token is not the most efficient of all processes, so you will find alternative solutions where you end up with a dedicated machine in a server rack. This all makes it a no-go for a low or zero budget situation. It also means that for just *printing*, *viewing* or *storing* purposes signing doesn't make that much sense: it only adds overhead.

One can argue that signing is not that robust anyway. Just like we can add a signature to a file, so can anybody. It's all about trust. When a byte in a pdf file is changed validation fails anyway so that is already a signal; we don't need to verify the certificate for that. And it's not that hard to let a user upload a file to the origin and let it validate there where the private key is known. But wait, isn't it more convenient to do that without uploading? Sure, but here are some pitfalls. First of all, who knows if a certificate is still valid? An organization has to spend quite some money on it yearly. And (even root) certificates expire so in the end the document refers to something invalid anyway, which effectively makes the document expire after some time. Saving documents and providing them again might be cheaper and also has the advantage of archiving. For long term archiving signing makes little sense anyway (expiration, cracking).

So why do we bother to add signing to ConTEXt? The answer is simple: user demand. Just like being forced to use some pdf standard, users can be forced to comply with what the organization prescribes with respect to signing, even when in the end it's just a demand, and nothing is actually done with it. So the main question is: after showing that it can be done, what eventually happens; how does the workflow look? It's comparable to tagging: it is sometimes demanded, but after that the lack of useful tools make it just a box to be ticked.

There are other alternatives to making users feel good about a document: provide a printed copy, keep the original someplace for downloading, maybe make it possible to regenerate a document from source, maybe even provide the resource. Generate a string hash and keep that available alongside the original. In the end it is all about trust, indeed.

Let's end on a positive note. Getting to know what has to end up in the file is not that trivial and as with much on the Internet, looking for solutions quickly brings you to a subset of partial and sometimes confusing answers and solutions. This is why in the end I decided to just look in the code base of `openssl` that

comes with examples and eventually one can sort out something not too complex. One of the interesting observations was that the binary blob is a structured key/value sequence using technology from decades ago (1984), when data had to be transferred reliably between architectures and programming languages: Abstract Syntax Notation One (ASN.1). It makes old TEXies feel young when old tools survive beyond the modern short lifespan of fancy web technologies. I might eventually spend some more time on this, just for the fun of it.

If you want to know more details: the official iso standard on pdf has some sections on the matter; a more comprehensive summary can be found in "Digital Signatures in a pdf, Adobe Systems Incorporated, May 2012". There is also "CDS Certificate Policy, Adobe Systems Incorporated, October 2005" but I suggest to ignore that one unless you're forced to implement the more expensive route.

Some final words on the mentioned formats. For printing and storage this feature is not needed. Nor for regular viewing, because users probably don't care that much if a manual or book is signed and it's unlikely that certificates last that long (or stay secure for that matter). But it might make sense for distributing documents with some legal meaning in the short term. In that perspective having this feature in ConTEXt makes most sense in specific workflows. But it doesn't hurt to know that TEX is still able to adapt itself to these situations.

Many thanks to Karl Berry for copy-editing this text!

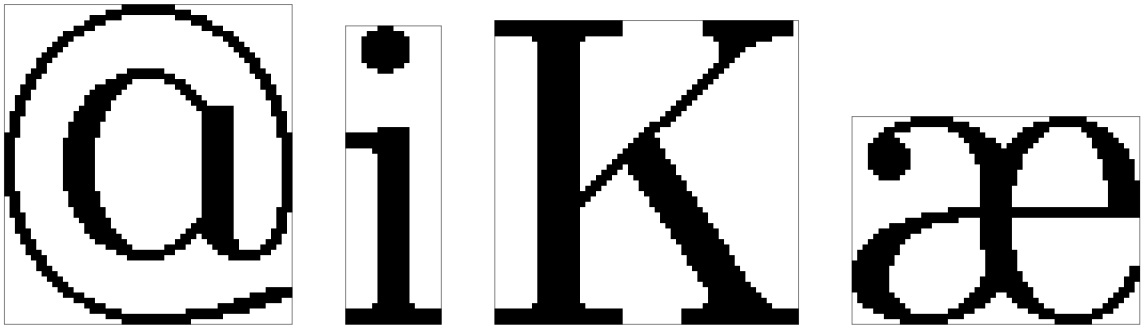
33 Bitmap fonts

This is a follow up on potrace-generated outlines from bitmaps (see preceding article, ``Unusual bitmaps''). Most of today's TEX users have probably never generated a document with bitmap fonts but back in the day we only had these, almost always Computer Modern. Because we don't use bitmap fonts in ConTEXt there is no need to support so-called pk (bitmap) fonts in the backend, but, because we did support them in MkIV, we still have it in LMTX. After all, what is TEX without bitmap fonts? We also owe it to Don Knuth.

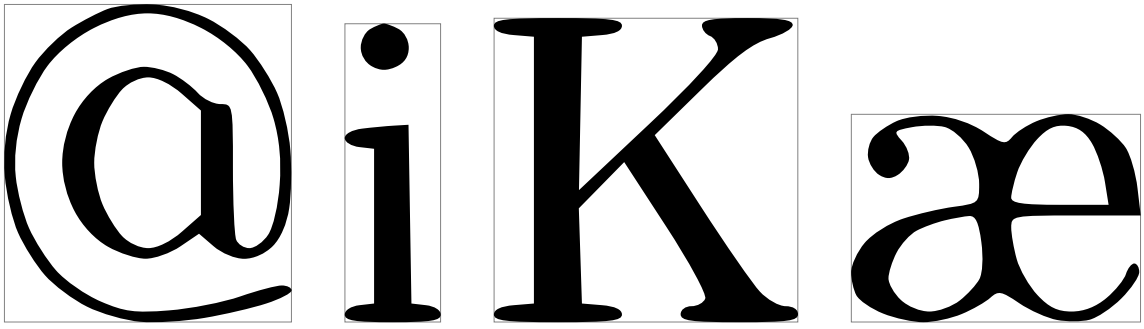
It is a bit of challenge to load a traditional eight bit font using only a tfm file because in ConTEXt we map everything to Unicode. Regular Type1 fonts are still supported, although they are declared obsolete, and vendors have moved on to OpenType fonts. When we define a font that has Type1 resources, we load it as if it were an OpenType font. These eight bit fonts thus become wide (up to 64K glyphs) fonts internally. Normally we consult the afm and pfb files and leave the tfm files, if present at all, for what they are. You can think of runtime `afmtotfm` conversion. An exception is the few traditional math fonts that we support: Antykwa, Iwona and Kurier; here the tfm files provide dimensions.

For the purpose of demonstrating what comes next it is enough to know that in principle one can still mess around with tfm values, either outline or bitmap, and that encoding files play a role in mapping them onto Unicode. We never set out to do something like this, but, because we had the loaders available anyway, some quick (few line) experiments of passing pk bitmaps to the MetaPost potrace helpers we got curious.

If you've run into an old TEX document on the web you might have noticed bitmap fonts being used. Often these are of a relatively low resolution. The reason that one never noticed that in print is that, when read on screen, we see glyphs large and can even zoom in, while in print they are seen small. When larger glyphs are used (say in a title) the scaled glyphs are actually different bitmaps: they have the same resolution as the smaller ones but more pixels because they are larger. Take these characters at 600 dpi, scaled up from their original 10 point size:



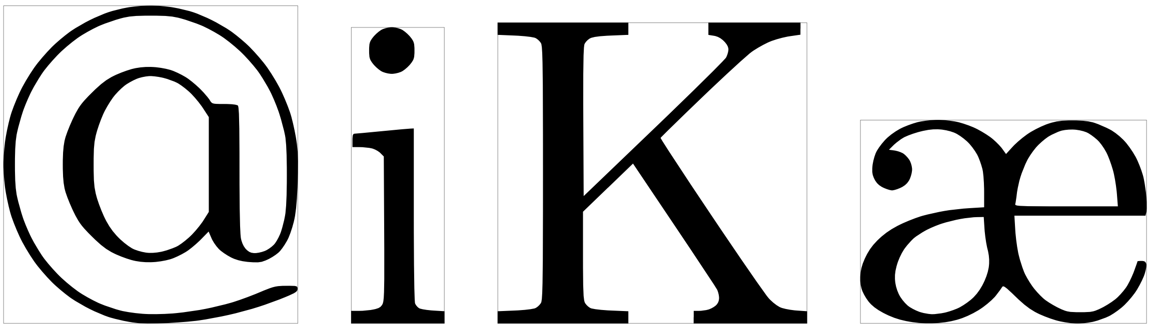
How do these patterns translate into an outline? For this we use the potrace library that we have available in LuaMetaTEX and interfaced to MetaFun in ConTEXt (as discussed in the companion article), although for fonts we follow a more direct route.



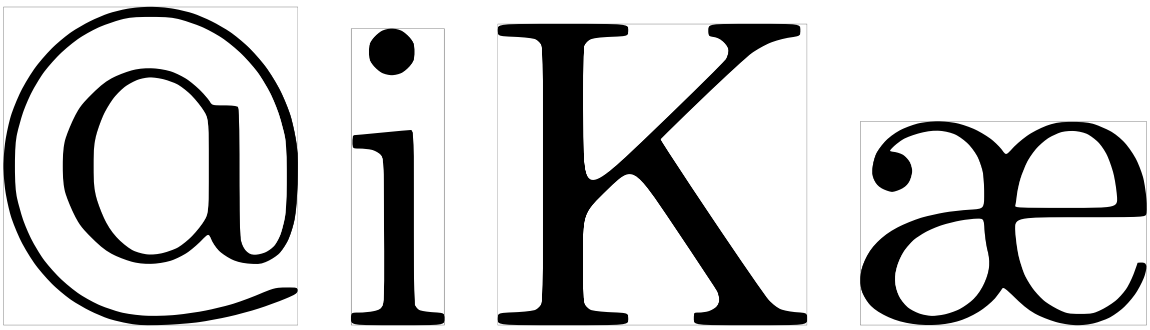
So why do these glyphs look somewhat different (smoother) from a normal outline Computer Modern? This is because the 600 may sound like a lot but actually isn't. Distributed over an inch we have 600 pixels on 25.4 mm so roughly one pixel per 0.05 mm, which might be acceptable in a small print but not when scaled. Here is how an 'm' in 600 dpi pixels is coded (64 by 37 pixels):

```
0000001111110000000011111111000000000000000011111111000000000000
1111111111110000001111111111110000000000011111111111100000000000
1111111111110000111000000011111100000000111000000011111100000000
1111111111110001110000000000111111000011100000000001111110000000
0000111111110011100000000000111111000111000000000001111110000000
0000001111101110000000000000111110011100000000000001111110000000
0000001111111100000000000000111110111000000000000001111110000000
0000001111111000000000000000011111110000000000000000111111000000
0000001111111000000000000000011111110000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000001111110000000000000000011111100000000000000000111111000000
0000011111111000000000000000111111110000000000000001111111100000
1111111111111111110000011111111111111111100000111111111111111111
1111111111111111110000011111111111111111100000111111111111111111
1111111111111111110000011111111111111111100000111111111111111111
```

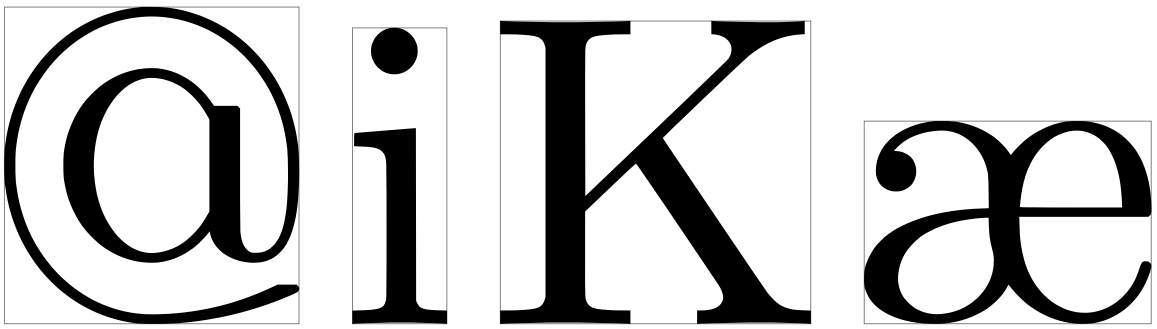
In the days of sending bitmaps to printers, possibly wrapped in a PostScript file, one could often recognize the characters from blobs like this. You might also remember some of that line printer art. At any rate, it shows that 600 dots per inch is much less than it sounds. A decent 2400 dpi (dots per inch) is more normal these days for printing on presses (with ink) but although there are 1200 dpi laser printers, 600 became the norm. After all, toner particles are not that small. For quite a while the authors used high speed OCE low temperature toner printer (first 508 dpi, later 600 dpi) and these were visually superior to most of what the competition had — but OCE never quite managed to do the same in color (only in lab testing, not reaching the market). Nowadays we use HP full width high speed inkjet printers that give a rather good quality full color experience at 600 dpi. This is what a scaled-up 2400 dpi bitmap look like:



This already looks more crisp although we can go back to the smoother variant by setting a higher threshold in potrace:



We can go higher. In the next rendering we use 7200 dpi bitmaps and now we see some details that didn't show up before. Keep in mind that subtle details might not be noticed in 10 point running text. Are the "ends of the at and e cut?"



The top of the 'K' now has a little dent (likely not visible except with magnification). If you ever viewed a document with Latin Modern outlines you might recognize this. It is a side effect of outlines having that information while a bitmap needs to get the exact bits, which won't happen if such a dent stays beyond the pixel threshold. We now are ready for some real text. We define two font features to load bitmap and outlines, respectively:

```
\definefontfeature[whateverpk][default][reencode=ontarget-cmr.enc,bitmap=pk]
\definefontfeature[whateverpt][default][reencode=ontarget-cmr.enc,bitmap=outline]
```

and some colors:

```
\definecolor[pkcolor][r=1,t=.5,a=1]
\definecolor[ptcolor][b=1,t=.5,a=1]
```

We use the following three font definitions in three overlaid examples (figure 33.1). The results are close enough to justify a closer look at the possibilities.

```
\definefont[PKdemoA][file:lmroman10-regular.otf*default sa 1.2]
\definefont[PKdemoB][file:ontarget-cmr10.tfm*whateverpk sa 1.2]
\definefont[PKdemoC][file:ontarget-cmr10.tfm*whateverpt sa 1.2]
```

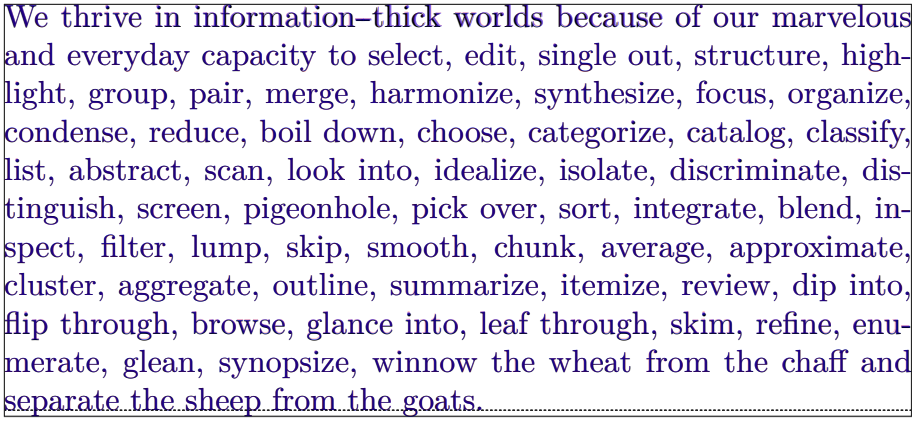


Figure 33.1 Overlayed regular

In figure 33.2 we see from top to bottom: Latin Modern OpenType outlines, a bitmap Computer Modern and a potraced Computer Modern. The fourth line has the bitmap and potraced overlaid. Figure 33.3 shows a small portion of the last one as seen on screen. Blown up, some drift is seen but so far we didn't find a way to get rid of it. Some is due to the way glyph streams get rendered and synchronized (after spacing, for instance).

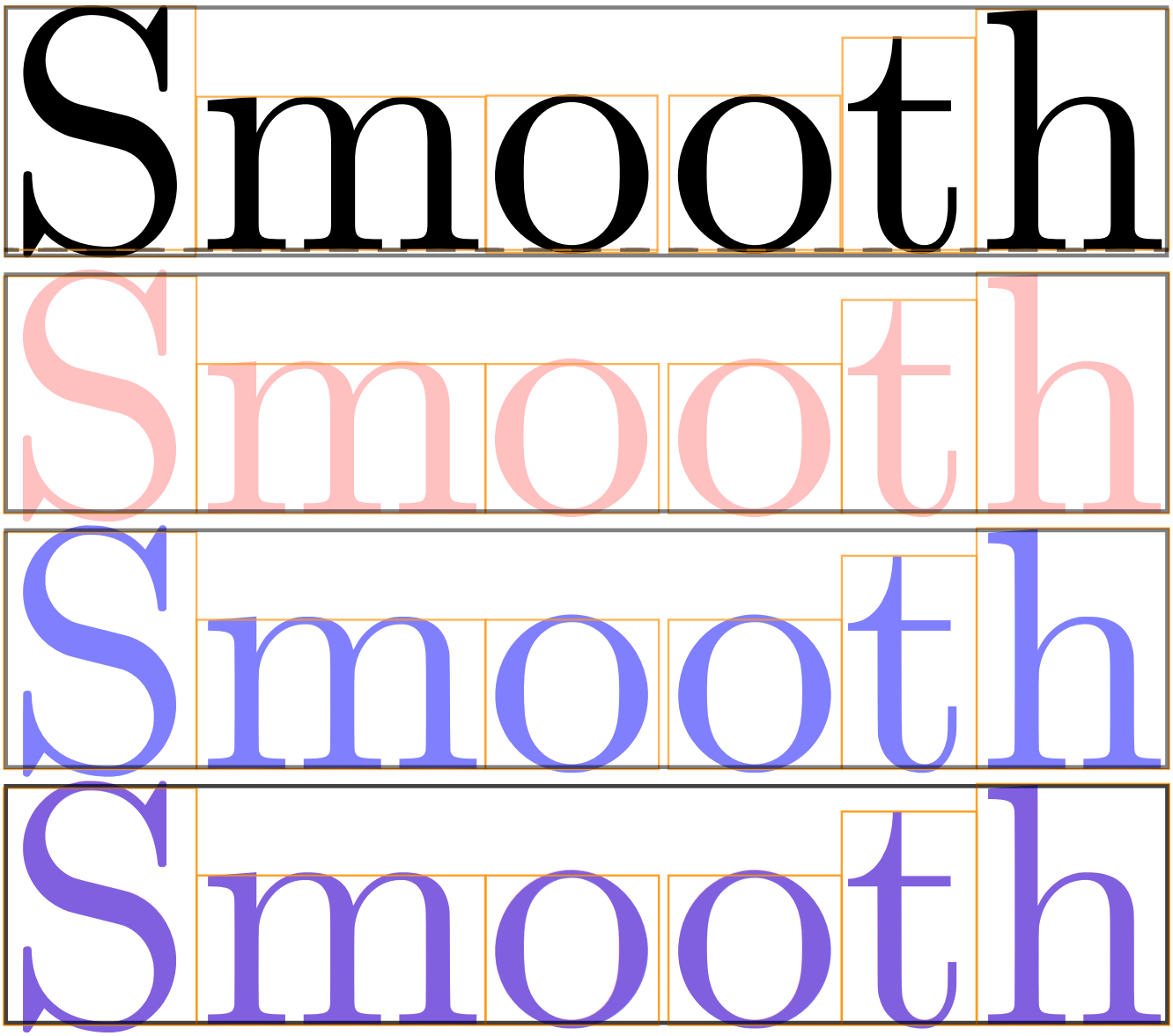


Figure 33.2 Bitmap and potraced compared: OpenType, pk, potraced & overlayed.

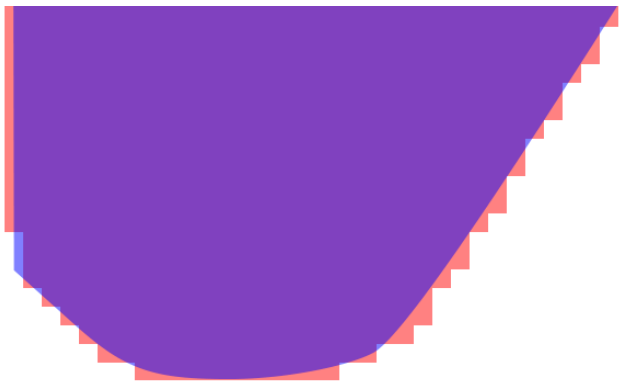


Figure 33.3 An enlarged clip of the overlay.

When not overlaid we get this for a normal Latin Modern Regular:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into,

idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

And this for a Computer Modern Roman pk bitmap:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

The same Computer Modern Roman outlined by potrace gives this:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Because these are outlines we can actually scale them nicely with `\glyphscale 800` here:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We can additionally scale an extra `\glyphxscale 1200`. This can of course also be done with bitmaps but outlines are a safer bet.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

The conversion happens in the backend and can have some impact on the runtime but we cache the outlines, so a subsequent run is faster. Of course outlines are more efficient than bitmaps in terms of bytes and viewers also tend to render them better than bitmaps, which, especially at a lower resolution, can look pretty bad (in some viewers).

One might wonder if bitmaps are worse than outlines. When we use a reasonable resolution there is no need to generate more than one size, and in ConTEXt we assume this anyway because we scale all fonts to

10 big points and scale that shared instance on demand. The 600 dpi ‘m’ that we showed has 64 pixels in the horizontal direction. So, 75 of these (a line of text) need 4800 4–8 display. Unfortunately viewers still render a bitmap font somewhat badly.

Let’s dive a little into why bitmaps might render suboptimally in pdf viewers. Consider these three lines typeset in our three fonts:

```
\PKdemoA Smooth \vskip.1ex
\PKdemoB Smooth \vskip.1ex
\PKdemoC Smooth \vskip.1ex
```

These ‘Smooth’ lines (this time in 12pt) end up in the pdf file as follows :

```
BT
/F1 10 Tf
1.195514 0 0 1.195514 0 30.316793 Tm [<000100020003>-28<000300040005>] TJ
/F2 10 Tf
1.195514 0 0 1.195514 0 15.415656 Tm [<010203>-28<030405>] TJ
/F3 10 Tf
1.195514 0 0 1.195514 0 0.514763 Tm [<010203>-28<030405>] TJ
ET
```

This code first switches to font `/F1`, a wide outline font so we have four-byte indices (in angle brackets). Next we trigger `/F2`, the bitmap variant and finally the potraced `/F3`; these are both Type3 fonts so they get two-byte indices. We don’t scale except to the 10 big points design size. After such a switch comes lines of text and there we do scale, here by 1.195514 in both directions. We’re slightly off 1.2 because the pdf font system (by tradition) is set up in PostScript (big) points so we need to scale up a little from 12pt to 12bp. Scaling an outline is translated (in the end) to some factor and the renderer can keep the device into account when it comes to rounding. With bitmaps it’s different, because these are not mathematically-defined fonts, but some image that gets scaled. This can introduce the first inaccuracy. An inline bitmap in pdf is given between `ID` and `EI` operators, as demonstrated in the next two charproc entries for the Type3 bitmap font (reformatted and abridged to save space):

```
21 0 obj
<< /Length 40708 >>
stream
556 0 55 -22 498 703 d1
q
442 0 0 725 55 -22 cm
BI
  /W 442 /H 725 /IM true /BPC 1 /D [1 0]
ID ...bytes...
EI
Q
endstream
endobj
```

Watch the different in length:

```
22 0 obj
<< /Length 42784 >>
stream
```

```
833 0 33 0 809 440 d1
q
775 0 0 440 33 0 cm
BI
  /W 775 /H 440 /IM true /BPC 1 /D [1 0]
ID ...bytes...
EI
Q
endstream
endobj
```

In contrast, a character in the potraced outline font looks like this:

```
31 0 obj
<< /Length 3678 >>
stream
556 0 55 -22 498 703 d1
q
1 0 0 1 55 -22 cm
172 723.960456969 m 144.844454411 720.63823631 ...
Q
endstream
endobj
```

The length is much smaller and the outline shape is just a sequence of moveto (`m`), lineto (`l`) and curveto (`c`) operators mixed with numbers.

```
32 0 obj
<< /Length 4260 >>
stream
833 0 33 0 809 440 d1
q
1 0 0 1 33 0 cm
68.5 434.441743787 m 32.2 431.509475939 1.9375 ...
Q
endstream
endobj
```

Experiments demonstrated that it's better to use rounded widths because otherwise (at least in Sumatra) we get some accumulated drift. The bitmap variants have a transform matrix like this:

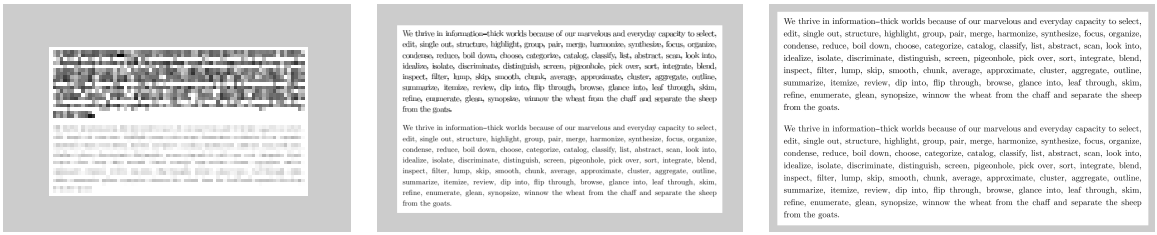
```
442 0 0 725 55 -22 cm
775 0 0 440 33   0 cm
```

and the potraced outlines have:

```
1 0 0 1 55 -22 cm
1 0 0 1 33   0 cm
```

This means that a bitmap again gets scaled, luckily by an integer, but still there is some inaccuracy. In the end, we get the bits put on screen and especially at small scales we end up with artifacts in positioning and scaling. Where the font renderer is optimized for (indeed) rendering fonts, the bitmap renderer isn't. In

figure 33.4 we see bitmaps being rendered bolder when they become smaller, because in the end, even at high resolutions, we're not talking pixels but bits (that can occupy multiple pixels). Outline fonts talk pixels, bitmap fonts speak in bits.



smallest small normal

Figure 33.4 Three zoom levels compared.

We haven't yet discussed how we got the bitmaps that we used. You might have noticed in the examples that there are some differences with the outline when it comes to dimensions. This is partly due to the fact that where Latin Modern is an OpenType font with no limits to dimensions, Computer Modern has to accommodate the limited number of heights, depths and widths that the tfm format permits. Think of arbitrary values of height compared to categories of height.

When you generate a bitmap you rely on scripts that do the work and these work together with so-called printer modes as defined in the MetaFont file `modes.mf` (https://ctan.org/pkg/modes). These modes are for printers which means that there can be compensation going on: rounding up or down of points exceeding bounding box edges, the size of printer pixels (toner, ink) and accuracy of positioning them, etc. In our case, when we went for 8000 dpi we ended up with a device that did more compensation than needed. Better is to investigate time in figuring out how to control the machinery to cook up (maybe) 7200 dpi bitmaps because down the inclusion route there is some division by 72. If we decide to play a bit more, we might as well first figure out how to control the bitmap generation and see if we can come up with this 7200 resolution. Not only does it divide nicely by 72 (for display) but also by 600 (for the average printer). To what extent that matters is to be seen.

A bitmap font (instance) is generated by MetaFont and driven by a (printer) mode defined in `modes.mf`. We added this one:

```
mode_def potrace =
    mode_param (pixels_per_inch, 2 * 3600) ;
    mode_param (blacker, 0) ;
    mode_param (fillin, 0) ;
    mode_param (o_correction, 1) ;
    mode_common_setup_ ;
enddef;
```

The `2 * 3600` is a trick to get around MetaFont maxing out at 4096 but internally being capable to deal with larger numbers. Of course we forgot to run `fmtutil-sys --byfmt mf` which is needed to get these modes in the format file, but eventually we managed to generate the 7200 dpi pk file for `cmr10`. Generating is easiest done with pdfTEX with `\pdfmapfile {}`, which wipes the mapping to a Type1 file.

We mentioned using MetaPost in our first attempts to get an idea how well potrace can vectorize the bitmaps. Here is how that is done:

```
\startluacode
    local f = fonts.handlers.tfm.readers.loadpk("cmr10.pk")
```

```
    if f then
        local g = f.glyphs[string.byte("R")]
        if g then
            local b = fonts.handlers.tfm.readers.showpk(g)
            potrace.setbitmap("demo",b)
        end
    end
\stopluacode

\startMPpage[offset=1ts]
    draw lmt_potraced [
        stringname = name,
        value      = "1",
    ];
\stopMPpage
```

In figure 33.5 we demonstrate three such renderings. Watch how the number of points increases as the shape gets better. In figure 33.6 we show the potraced variant alongside the OpenType outline. Left is the default potrace output, next comes the regular OpenType (here cff) outline, and at the right we see two potraced results, both with `optimize = true` passed; the first has the default tolerance of 0.2 and the second uses 0.5 and thereby has less points. For sure the middle one has less points which is nice, but the potrace ones are not that excessive so we can live with it. We've run into cases (especially math) where the regular outlines are also not perfect. Most will go unnoticed anyway given the small size at which glyphs are normally rendered. When you look closely at the rightmost output you'll notice that the bend in the stems makes for more points which is an indication that the MetaFont output might actually be more subtle. In figure 33.7 we also show the controls and you see subtle differences in the angles there. Also note that when there are no lines that indicates that there is likely a lineto instead of a curveto.

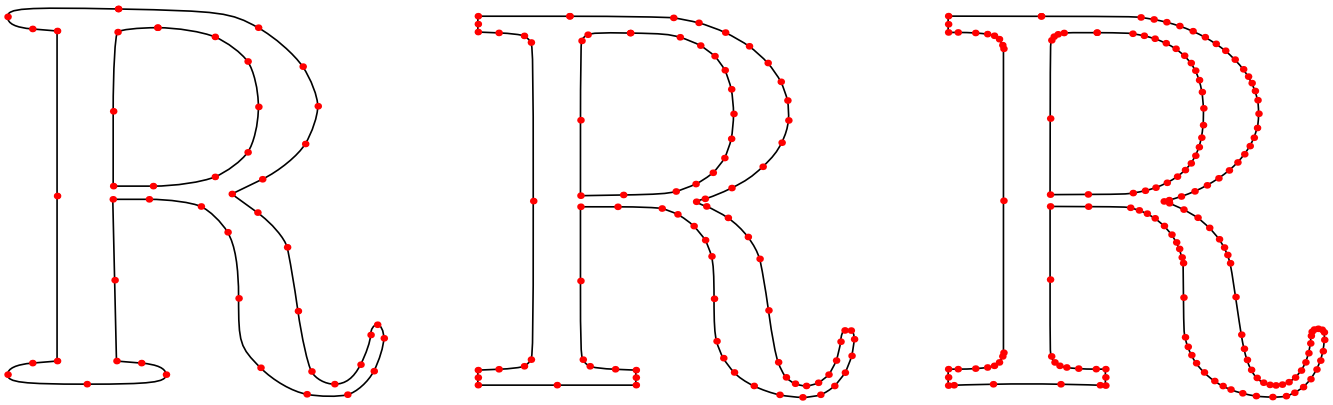


Figure 33.5 Using MetaPost for analysis.

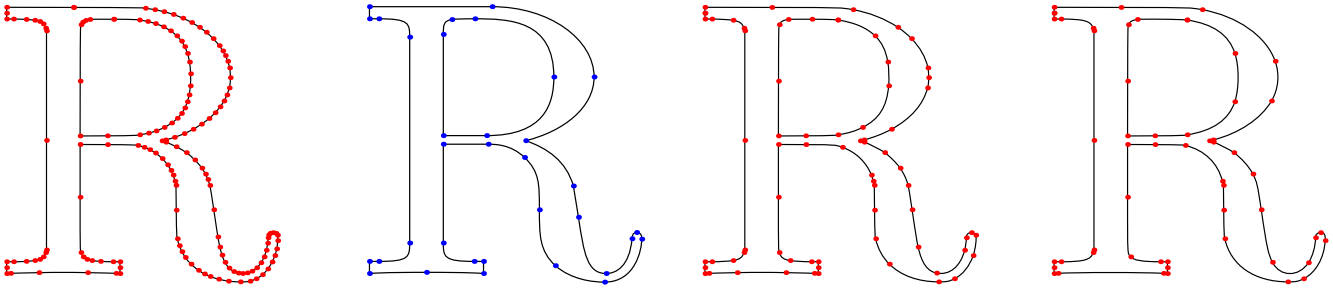


Figure 33.6 Comparing potraced bitmaps with Type1.

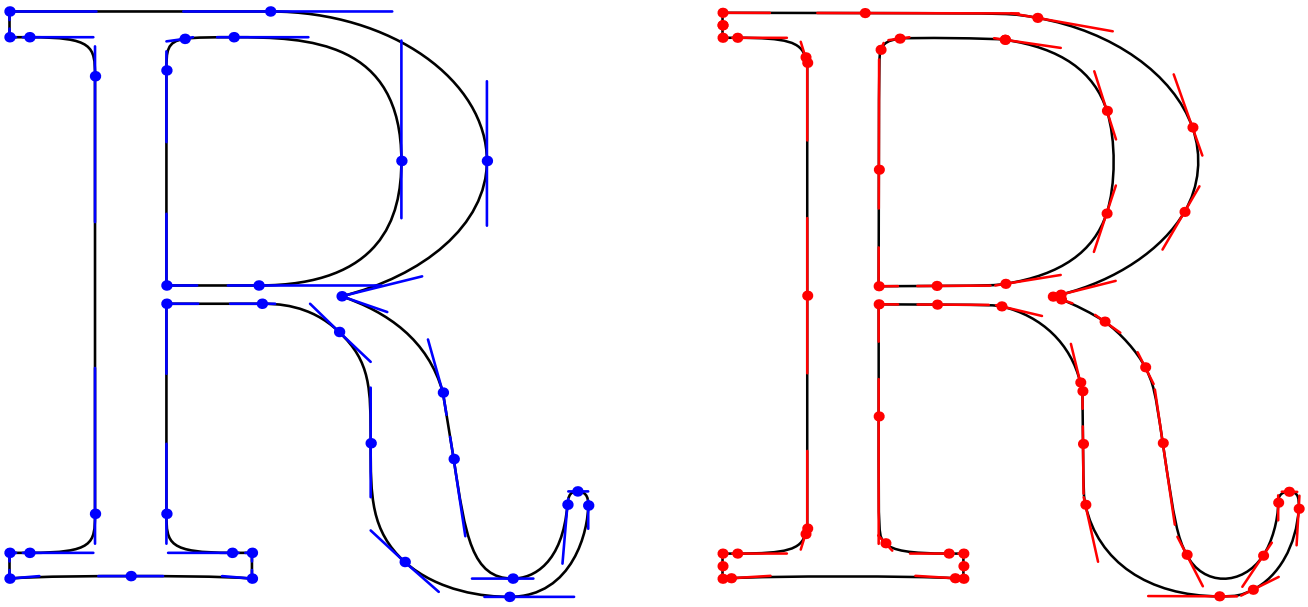


Figure 33.7 Comparing control points.

If you like the look of these shapes, take a look at Volume E of Don Knuth's "Computers & Typesetting" series. An incredible amount of work went into the details of the fonts and you'll run into brackets, beaks, crisps, notches, slabs, juts, dishes and more elements and parameters that are used. For instance the serifs as seen in the 'R' are actually made programmatically (so that they can be discarded in the sans shapes). If you check out the proof sheet of the 'R' you will only see the basic points that describe the character, not the points we see above, after conversion to (any kind of) outline.

So now that we can turn a pk into a proper outline we're done, right? Well, not entirely. Because we have relatively simple shapes (moveto, lineto and curveto) we can directly go to cff and avoid the MetaPost to Type3 conversions. Because we already can load (and adapt) cff outlines it is not that complicated to do the reverse. The backend already can include them so we can also borrow code there.

When going from potrace output to cff, we need a high resolution, so we started out again with 7200 dpi. Although in the end we were quite satisfied with the results, we tested with 20000 dpi and thought it looks even better than the Latin Modern successor (although one can argue about it). Some first experiments showed that it was doable but it actually took a whole day to (test and) decide how to get better results. For instance, MetaPost uses floats while a renderer uses integers, so we run into rounding issues if we delegate that (although we can include floats in cff, it is not a success). When overlaying the MetaPost output and the cff shapes the latter was quite disappointing: weird protrusions and bubbles. Of course one may doubt the implementation, but double and triple checking showed that we use the same numbers.

The results improved a lot when the results from potrace were multiplied by 10 (or 20) effectively giving very huge glyphs (on a 10000 unit canvas) and in the page stream scaling down by that factor. By then using rounded values we got enough precision to get the cff results close to the (always) high quality MetaPost rendering. In figure 33.8 we can see how close they became.

The next question is, how do we control this: pk, MetaPost Type3 or native cff? Even more challenging is that we had wanted to use different resolutions, and mix these three methods, if only because we want to be able to experiment and document the mix. That means that it has to be available not only in the font feature mechanism, which is rather trivial, but also in the backend, which then involves a font with the same name (say `CMR10`) to be rendered differently, so we need different handlers, distinctive caching of streams, etc. In the end we got there. It is even possible to mix variants with different potrace parameters in one document.

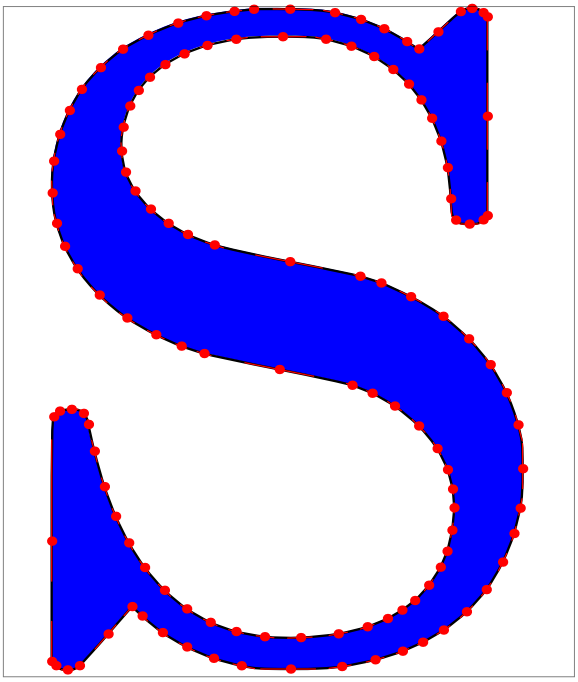


Figure 33.8 A MetaPost rendering on top of the cff counterpart

We start with cff definitions. In figure 33.9 we show three resolutions overlaid, with the 7200 dpi variant below. In figure 33.10 we show the default and optimized (fewer points) traces.

```
\definefontfeature[CMRCFF][reencode=ontarget-cmr.enc,bitmap=cff]

\definefontfeature[MyFontCffA][default,CMRCFF][resolution=7200]
\definefontfeature[MyFontCffB][default,CMRCFF][resolution=2400]
\definefontfeature[MyFontCffC][default,CMRCFF][resolution=600]
\definefontfeature[MyFontCffD][default,CMRCFF][resolution=7200,potrace={optimize=true}]
```

The overlays look fuzzy, demonstrating the need for high resolutions. Font definitions are done as follows; we only show one definition:

```
\definefont[MyFont][file:ontarget-cmr10.tfm*MyFontCffA]
```

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 33.9 The 7200, 2400 and 600 cff variants overlayed.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 33.10 The normal 7200 and optimized 7200 dpi variants

```
\definefontfeature[CMRPK][reencode=ontarget-cmr.enc,bitmap=pk]

\definefontfeature[MyFontBitmapA][default,CMRPK][resolution=7200]
\definefontfeature[MyFontBitmapB][default,CMRPK][resolution=2400]
\definefontfeature[MyFontBitmapC][default,CMRPK][resolution=600]
```

These definitions are used in figure 33.11. The differences aren't immediately apparent in small print but zoom in (if you're online) and you'll understand why we need more than 600 dpi to feel comfortable.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 33.11 The 7200, 2400 and 600 pk variants overlayed.

```
\definefontfeature[CMROUTLINE][reencode=ontarget-cmr.enc,bitmap=outline]

\definefontfeature[MyFontOutlineA][default,CMROUTLINE][resolution=7200]
\definefontfeature[MyFontOutlineB][default,CMROUTLINE][resolution=2400]
\definefontfeature[MyFontOutlineC][default,CMROUTLINE][resolution=600]
```

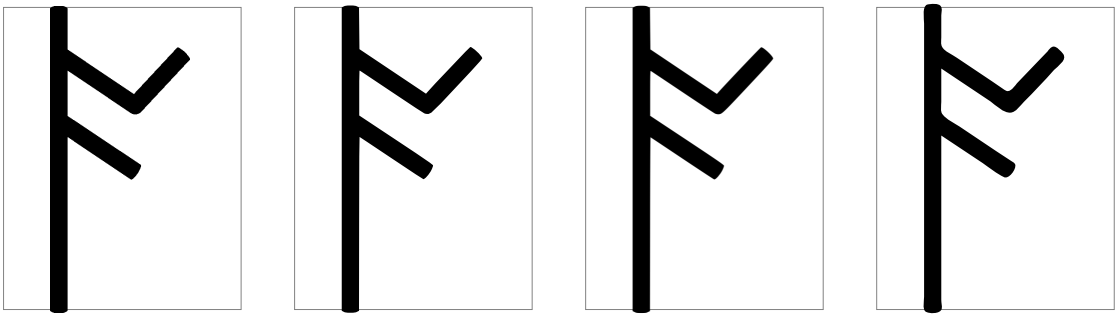
Finally we show the MetaPost-generated threesome in figure 33.12 and these look quite reasonable. One thing to keep in mind when wrapping shapes into a Type3 font is that one has to make sure that color keeps working.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

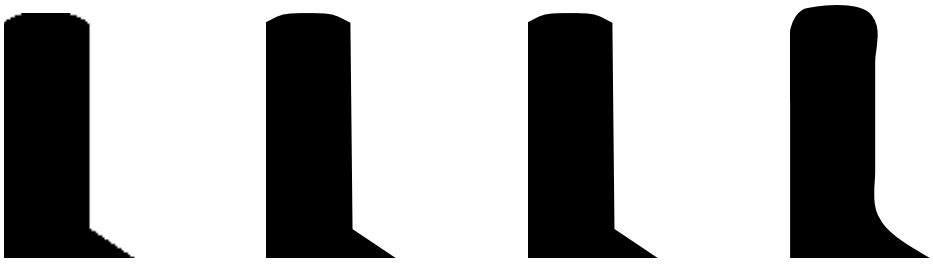
Figure 33.12 The 7200, 2400 and 600 MetaPost variants overlayed.

So, how useful is all this? Maybe it's time for a revival of MetaFont. Or maybe we can get some old designs out of the archives where they got tagged obsolete and use them again. Or maybe it's just for the fun of it. We started out with pk bitmaps that we need to support anyway. Next we were curious how well a traced outline would look, and for that using a MetaPost Type3 font makes sense. Then we took the challenge to turn potrace output into a cff OpenType font. We could now make a whole tool chain but it makes little sense: we can do it in ConTEXt, so we're fine, and we don't expect widespread usage outside the TEX ecosystem. Next on the agenda is to locate some interesting MetaFonts and see if they can be put to use. In case you wonder how these eight bit fonts fit into the Unicode ecosystem, don't worry, we can use the virtual font mechanism to hook shapes into existing fonts (which is what we do anyway) or create combined fonts. We can even make variable fonts using MetaFont, but for that we need to become experienced designers first. As a teaser we show a character from the runes font by Carl-Gustav Werner in figures 33.13 and 33.14. Mikael, knowing the author, promised to come up with a proper encoding for that one, so stay tuned.



Type3: pk Type3: MetaPost OpenType: cff Type1: pfb

Figure 33.13 A simple test with one of the allrunes fonts. Left is using the pk at a 7200 resolution. Next is the potraced 7200 pk original outline. Third from left is the generated cff. Right is the shipped pfb.



Type3: pk Type3: MetaPost OpenType: cff Type1: pfb

Figure 33.14 Enlarged clips the four variants in figure 33.13.

34 Meaningful math

Hans Hagen
Mikael Sundqvist

34.1 Introduction

In this article we're going to discuss math in the perspective of accessibility. Although ConTEXt already supports tagging in pdf for quite a while, that specific kind of accessibility never took off, if only because very few viewers did anything useful with it. However, with universities introducing (whatever) validating features for documents pushed into the systems used for teaching, there was no way around picking up this thread. We start with some reflections about how we got here and then move on to some examples of how we deal with this in LMTX. This project is part of a larger effort to get even better typeset math out of TEX so we could benefit from some new features in the engine, even if they were never added with accessibility in mind.

34.2 How about MATHML

Hans still remembers one of the first TEX conferences that he attended where some evangelist from a large, in the meantime extinct, hardware company tried to convince the audience that something great had surfaced: xml. In its wake came MathML, as far as we understood endorsed by large mainstream scientific publishers, some of which also disappeared in the meantime.

He also remembers a user group meeting where, when showing some MathML processing, responses were of the kind "Why do you use TEX at all if you want to use MathML (or xml)" or even more curious "Why do you use TEX if isn't for math?" Sometimes such comments are a way to put ConTEXt (users) in their own category. Remarks of a similar kind are "One should not use ConTEXt because it depends on some scripts". The fact that scripts are used to (for instance) sort an index, process a bibliography or just manage the multiple run process (also think of MetaPost graphics and position tracking in dvi in former times) was seen as bad. Of course in the meantime many such features have proven to be useful, which is illustrated by the fact that other macro packages also have to deal with MathML and seem to use scripts. We mention this because decades of writing ConTEXt code has taught us not to bother too much about what is said or happens outside our scope.

Because we were actually confronted by xml (or its precursor sgml) it is no surprise that in ConTEXt already pretty soon the angle bracket dominated format was supported and then of course MathML support is kind of easy to add to the portfolio. Although the way we deal with xml is more sophisticated in MkIV, and therefore the same in MkXL aka LMTX, the kind of things we do are the same.

34.3 Practical use

At Pragma ADE we were never that thrilled by presentation MathML but content MathML has a certain charm and was actually more reliable when it came to rendering, if only because soon the usage pattern of the presentation variant became a wildcard. Where dealing with content MathML is kind of fun, handling presentation MathML is less so. For decades we've been involved in a project where the whole spectrum of Dutch secondary math education is covered. The content is mostly developed on a voluntary basis so for us it was mainly a way to experiment with large scale rendering of xml mixed with math. The target was pdf as well as the web, but we were not responsible for that part (although we can generate a web site if needed). However,

as we often found out, the (limitations of) the web (html) influenced the xml coding in a negative way but we leave that aside. It just makes the source code less attractive over time.

If we use that project as example, we can only conclude that MathML often looks like it failed to mature. We're talking about 30.000 files with over 400.000 math formulas but it being secondary school math they differ in complexity. The books are generated on demand and can differ in content and detail. There are of course also lots of images involved. A single run can involve thousands of files that are loaded, combined, filtered and typeset and in spite of what one might expect processing speed is quite comfortable (not that much different from the average ConTEXt run). Static and (screen optimized) interactive pdf files, are path of the package. Some schools even generate their own variants, given that they are willing to install th e production system and invest some time in learning how to mess with the content and assembly files.

The disappointment with MathML has a reason. We started very optimistic with content MathML but because that demanded transformations for the html variant we moved to presentation MathML. Then there was some involvement from OpenMath participants and all was moved to that format because a related output demanded it. At that time ConTEXt was able to process all three variants so for book production that was not really a problem. Nevertheless another change was needed, this time because browser support was (to put it mildly) fragile and with again a flirt with presentation MathML we finally ended up with AsciiMath which has its own peculiarities but at least was supported via MathJax. So that's where we are now: of the over 400.000 formulas 52.000 are presentation MathML and 360.000 are AsciiMath.

34.4 It comes and goes

Quite often when it comes to typesetting, the typewriter approach is taken as reference. As a consequence plenty of simple source tagging methods and fast rendering tools have showed up but it is only a matter of time before one hits limitations and coding becomes more complex due to handling border cases and exceptions, and rendering becomes slower or maybe even cumbersome because a complex layout is well, just complex to realize due to conflicting demands. This is why TEX macro packages are large and complex too. So, when discussing for instance pdf documents (other than straightforward texts like novels or simple reports) in the perspective of the internet, one tends to forget that html is (at least initially) meant for viewing in browsers where it can reflow and adapt. The more one tends to markup there, the less easy it becomes and one only has to look at the source and style sheets to see that in the end it all converges to the same level of complexity. What matters here is not so much the rendering backend, but more the availability of a clean source, one that could be used to satisfy different needs, but quite often that is not the case: there is no way to control users. If 'simple' is the starting point, then 'complex' can be pretty ugly. The same is true for math: the more complex it gets, the more likely it is that the source is a bit of a mess. It is for that reason that in ConTEXt we paid a lot of attention to help users keep the source clean. It is also why we spent years on upgrading the core TEX math engine.

Anyway, in the meantime MathML went from version 1 to version 2 and 3 and is now a hybrid of presentation, content and OpenMath with version 4 around the corner. Then there's also a light version showing up: MathML core. We can probably safely conclude that in the meantime MathJax is mostly used in browsers and it's a combination of JavaScript, css, TEX fonts that handles presentation MathMLand AsciiMath (or maybe even to some extent LATEX math) either by element or by parsing the content of elements. One observation is that recent versions of MathML add lots of attributes to get things right. A with svg as soon as markup and styling enters the otherwise reasonable structured input it might get worse in the end. Since we delegate a lot to the engine we don't need that level of detail so for now we just ignore most of it.

Some browsers did momentarily support MathML directly, just in order to drop it overnight or change the coverage. As we're talking of decades it clearly shows that it's a mess. It is therefore surprising (and impressive) that one can actually render math in web pages as for instance Wikipedia demonstrates. One can

wonder what MathML core brings to this, especially as it expects browser support, while browser reputation is not that good. There is no reason to assume that the ever changing web landscape with a small number of implementations of supposedly generic formats will be better in the future. Maybe the problem is just complex. Anyway, it made me somewhat indifferent to the matter: we dealt with in the past, probably can deal with it in the future (if only by ignoring what never will be generally supported anyway). What is worrying, is that the existing already rather limited presentation MathML standard dropped (according to the Mozilla documentation) `mfenced`, `menclose` and some more, and on the other hand kept the redundant `sqrt`, didn't add some missing elements (there's more than operators, identifiers and numbers), introduced some weird replacements for the enclosed content (with school math as excuse – well there's more to it than that).

34.5 Messing up PDF

A secondary element in this game is pdf that has some MathML related tagging on board. There we have the fences so that can hardly be dropped from the standard. Then there is a reasonable addition in the sense that one can embed the whole formula instead. But all this is done in the perspective of accessibility and getting that from typeset text is a bit of a challenge unless one renders a dedicated version suitable for reading out loud, and there is no reason not to have two variants of the same document: it might even make more sense than cooking up methods to use the (always suboptimal) tagged content. Within reasonable bounds we always supported tagging but we have to admit that we never really got much feed back on that and the reason is obvious: there has hardly been any support from viewers. It is all one big gamble and likely older tagged documents are not that useable anyway because one had to gamble about how to tag best. Also, validating applications (part of school platforms) don't know that well how to check and report on it anyway. That leaves us with guesswork and just trying to do the best we can: using common sense works out best.

One 'problem' is that ConTEXt users have never been involved in MathML discussions, pdf tagging and (if there ever were) discussions about math in the TEX engine, simply because early in TEX's history LATEX became the de facto standard of operating. In that perspective, looking from the ConTEXt end, it's not much different from looking at how Microsoft word or other ways of coding documents influence these standards. We just need to either adapt or ignore what happens and the latter is often the best (given the mentioned instable situation).

And so, we just focus on the rendering, not caring much if it can be resembled in for instance MathML or tags. We're pretty sure that hardly any user wants to cut and paste a formula from a pdf document and accessibility can often be made better by better and/or different rendering (maybe combine with some robust tagging). The best we can do is to produce a clean output and make sure that enough information is in the file to (maybe) help someone who doesn't have access to the source.

34.6 What do we want?

When we talk about accessibility for visually disabled readers, there is not much we can say about it simply because we cannot experience the same. The closest we can come to it is by listening to a document being read in (say) a podcast. One doesn't see the rendered output and depends on some software reader. Now, if accessibility has been taken into account, of course a dedicated pdf file with a layout that has no fancy rendering is best but what publisher is willing to provide two versions for that purpose? In some way we have a binary situation: if you can see the document, even when it has to be enlarged, there is little need for listening, and otherwise there is need for a nice layout. But in the end the single document demand will drive the curious mix of fancy layout and straightforward reading. But we fear that as long as big business, politics and money are involved we're unlikely to get anywhere close to a nice solution.

That said, the route we've chosen is serialization. When a user is forced to validate accessibility in some application and the verdict is negative, the first question is: "So, how should it be done then?" and of course there is seldom an answer to this. Adding tags to a pdf document is often enough to get into the green, but as said that tells nothing about the useability. We can embed the MathML representation and that actually satisfies some checkers. If we then add the serialized (read: spoken) math as actual text property the score can become positive there too.

Serialization started as a gimmick when we tried to satisfy demands and at first we actually produced a document where all formulas could be replaced by their 'spoken' variant. But then we realized that if someone has a visual handicap that might work out well but that it still is not much different from a typeset formula. Real accessibility would be better served with a more dedicated rendering of the whole document, not some partial serialization. We therefore decided to stick to the combination of embedded MathML and serialization in actual text annotations and just wait till viewers start supporting that. We do the best we can at our (rendering) end and leave it at that.

It must be said that when we started with this side track it was actually kind of fun. Compared to all we did with math (fonts, engine, additional and upgraded macros) this was a relatively simple activity, given that ConTEXt already had most on board and we could benefit from recently added features to the engine. It also made for a nice experiment with math dictionaries, something we had (and have) on the agenda anyway. Also, as with many mechanisms it might evolve over time, for instance because languages have different demands and currently we only cover English and Swedish. The tests are done with some real math books and the upcoming math manual. We're talking of thousands of inline and display formulas of different complexity here. For testing we have various tracing options which of course adds to the runtime, but generally speaking the overhead is quite acceptable, especially given the task at hand.

We started with some reflections on MathML as input and ended with MathML as output. We already had that for quite a while and could even embed it for tracing but now embedding is more integrated. The serialization is a nice addition to that so let's now explain a bit more about that.

34.7 What does it mean?

In TEX we often focus mainly on how to typeset math formulas and not so much about the meaning of those formulas. It should be clear that $\sqrt{x}$ stands for the square root of x, but many other symbols are used with more than one meaning. When we see a formula, we can often guess the intended meaning from the context, in particular if the author has used standard notation, introduced new notation, not used the same notation to mean several things, and kept the notation as simple as possible. There are, however, ambiguous cases.

We give one example where the situation might not be completely clear. If, in a manuscript on complex analysis, we meet the formula $\bar{z} \in \overline{U}$, we will likely interpret the first bar as the complex conjugate of the complex number z. But the meaning of $\overline{U}$ is perhaps less clear. The $\in$ hints that it should be a set. One standard notation implies that it denotes the closure of the set U. But it could also, in principle, mean the set of complex conjugate of the elements in the set U. Even if the bars over these characters look the same in the pdf file, it would be good if it was possible also to carry the meaning somehow.

If somebody copies the formula from the pdf they should get something sensible out of it when pasting it elsewhere. It is therefore important to work with the symbols predefined in Unicode math, and not to come up with their own weird symbols by clipping, rotating, or in other problematic ways combining symbols and perhaps also rules.

Unicode math contains a lot of symbols. Many of them are described with a name that rather says something about the shape than about how they are supposed to be used. Given that we are free to use whatever symbol

to denote anything, this is perhaps natural. But it is also problematic. Take $\otimes$ (its official name is CIRCLED TIMES), for example. It comes with two synonyms that tell a bit how it can be used as “tensor product” and as “vector pointing into page”. For the first usage the macro name `\otimes` has traditionally in TEX been attached to the symbol. But, as the synonym says, sometimes it also denotes a vector pointing into the page, and then one can question both the macro name and the binary operation class that is attached to it. If one wants to use this symbol in the latter meaning it is natural to define a new macro that typesets the symbol, with a matching class. Observe, however, that such a construct will not change the meaning for someone copying the symbol from the pdf. It will still be CIRCLED TIMES.

34.8 Accessibility, tagging

When it comes to accessibility, the situation becomes even more interesting. How shall a screen reader read the symbol $\otimes$? As “CIRCLED TIMES”, as “tensor product” or as “points into the page”? Or perhaps as “otimes”? ConTEXt has for a long time been able to tag documents that include mathematics and also export them to MathML. As of now, the formulas are transformed into some core MathML and included as attachments in the pdf file. Meaning easily gets lost in this conversion, so one can question how accessible the result actually is for a person who needs to have it read aloud. Moreover, the MathML itself, or the flavor of it, sometimes changes. For example, the `mfenced` element recently got deprecated, leading to compatibility issues for a lot of existing documents. This lack of stability makes it both difficult and demotivating to support tagging.

It can be useful to have the MathML if one wants to export and show the output on a web site. One shall remember, though, that the typeset math from ConTEXt that one gets in a pdf file is not in general equivalent (features differ) to the MathML produced, so it will not be perfect.

The example $\bar{z} \in \overline{U}$, discussed in the introduction comes out like this (we have replaced the math italic z and U so that they show below since they are not present in the monotype font we use)

```
<math>
 <mrow>
  <mover>
   <mi>z</mi>
   <mo>¯</mo>
  </mover>
  <mo>∈</mo>
  <mover>
   <mi>U</mi>
   <mo>¯</mo>
  </mover>
 </mrow>
</math>
```

Let us also mention that it is not easy to verify that the tagging actually works. At Lund university, where Mikael is working, the tool Ally (as a plugin to the Canvas LMS) is used to check the tagged pdf files, and it does usually mark tagged pdf files from ConTEXt as being perfectly tagged. But even here, things do change. At some point it was changed so that formulas was seen as attached images, and then it complained about lacking alternative texts. It is also an interesting fact that exporting a pdf file, verified as perfectly tagged, into sound (also possible in Canvas LMS), it does not read the formulas correctly, if at all.

34.9 Dictionaries

With the right mark up and choice of notation from the writer, it should be possible to have it read different things, depending on the context. We therefore came up with dictionaries. They make it possible for symbols to carry a meaning and context, in addition to the atom class. In fact, we shall think of it as something that is independent of the atom class. A Unicode character can thus have several instances, where different instances might belong to different groups. Of course the math class can also vary. Thus, for ⊗ it could be like this:

Symbol	Macro	Class	Group	Meaning
⊗	\tensorproduct	binary operator	binary operator	tensor product
⊗	\pointsintopage	ordinary	unary arithmetic	points into the page

The idea is then that the user can specify the groups used in a document. If one typesets a document on mathematical logic, one can load the groups that are attached to logic, and one will have these macros predefined, likely with the intended meaning. One can of course add or change the setup as well.

34.10 Formulas converted into text

One reason to introduce dictionaries with groups, in addition to atom classes, is that we can now use the label system in ConTEXt to attach to each symbol also a label that tells how it could be read out. The same has been done for various macros, and as a result we can convert formulas into "spoken mathematics", something that will be easily read by screen readers, since it is only text. Of course, given the amount of symbols and macros, we are not complete. In fact, we do not want to be complete either, and the reason is simple: We cannot know how various authors want their formulas to be spoken. So, what we have is merely a proof of concept, with a set of interpretations that covers many basic usages of commonly used symbols.

To get a hold of it, let us look at a few simple examples, where after each formula we show how it is interpreted in text.

```
\startformula
  1 + 2 = 3
\stopformula
```

$$1 + 2 = 3$$

[1] 1 plus 2 equals 3

```
\startformula
  3^2 + 4^2 = 5^2
\stopformula
```

$$3^2 + 4^2 = 5^2$$

[2] 3 squared plus 4 squared equals 5 squared

```
\startformula
  \frac{3}{6} = \frac{1}{2} = 1/2
\stopformula
```

$$\frac{3}{6} = \frac{1}{2} = 1/2$$

[3] the fraction of 3 and 6 equals the fraction of 1 and 2 equals 1 divided by 2

```
\startformula
  \sqrt{9} = 3
\stopformula
```

$$\sqrt{9} = 3$$

[4] the square root of 9 equals 3

```
\startformula
  \sin \left(\frac{\pi}{6}\right) = \frac{1}{2}
\stopformula
```

$$\sin\left(\frac{\pi}{6}\right) = \frac{1}{2}$$

[5] sin fenced the fraction of π and 6 end fenced equals the fraction of 1 and 2

```
\startformula
  \conjugate{1 + 2\ii} = 1 - 2\ii
\stopformula
```

$$\overline{1 + 2i} = 1 - 2i$$

[6] the conjugate of 1 plus 2 i equals 1 minus 2 i

```
\startformula
  \frac{1 + 2}{3 + 4} = \frac{3}{7}
\stopformula
```

$$\frac{1 + 2}{3 + 4} = \frac{3}{7}$$

[7] the fraction of 1 plus 2 and denominator 3 plus 4 end denominator equals the fraction of 3 and 7

34.11 Some difficulties

The process has really been trial and error. for sure there is space for improvements and variations, but we believe that the main structure is there. Different areas of mathematics come with different notations and different ways to interpret. So, if for example a logician wants to take this up, there is for sure some basic tuning needed before it works as expected.

One of the difficulties we encountered along the way was how to work with parentheses. When we write $a(b + c)$ we likely read it as "a times b plus c". But we cannot read it like that, since that could equally well be interpreted as $ab + c$. We need the parentheses to be interpreted as some group:

```
\startformula
  a(b + c)
\stopformula
```

$$a(b + c)$$

[8] a times group b plus c end group

On the other hand, when we write $f(x)$ it is likely that it shall be interpreted as "f of x" rather than "f times x".

```
\startformula
  f(x) \neq f\of(x)
\stopformula
```

$$f(x) \neq f(x) \quad [9]$$

[9] f times group x end group is not equal to f of group x end group

In addition to the `\of` to handle this case, we also introduced the possibility to declare glyphs as being functions. So, it is possible to do

```
  \registermathfunction[f]
```

and then leave out the `\of`. In fact, one of the main difficulties has been to control when the explicit "times" shall be there and when it shall not. There are several special cases; we have likely missed a few.

It is also possible to declare whole alphabets as being for example vectors or matrices. We can do

```
\registermathsymbol[default][en][lowercasebold][the vector]
```

and then use them as follows:

```
\startformula
  (\alpha + \beta) \mathbf{u} = \alpha \mathbf{u} + \beta \mathbf{u}
\stopformula
```

$$(\alpha + \beta)\mathbf{u} = \alpha\mathbf{u} + \beta\mathbf{u} \quad [10]$$

[10] group α plus β end group times the vector $\mathbf{u}$ equals α times the vector $\mathbf{u}$ plus β times the vector $\mathbf{u}$

34.12 A few more examples

We give a few more examples for you to ponder.

```
\startformula
  a__1(1 + x) + (1 + y)b__1 - a_2(1 + z) - (1 + u)b_2
\stopformula
```

$$a_1(1 + x) + (1 + y)b_1 - a_2(1 + z) - (1 + u)b_2 \quad [11]$$

[11] a with lower index 1 times group 1 plus x end group plus group 1 plus y end group times b with lower index 1 minus a with lower index 2 times group 1 plus z end group minus group 1 plus u end group times b with lower index 2

```
\startformula
  a__{0}.a__{1}\notimes a__{2} \ldots a__{n} \ldots
\stopformula
```

$$a_0.a_1 a_2 \ldots a_n \ldots \quad [12]$$

[12] a with lower index 0 . a with lower index 1 , a with lower index 2 , and so on, a with lower index n , and so on,

```
\startformula
```

```
  h'\of(x) \neq h'(x)
\stopformula
```

$$h'(x) \neq h'(x) \quad [13]$$

[13] h prime of group x end group is not equal to h prime of group x end group

```
\startformula
  s\of(1) = s\of(\set{0}) = \set{0} \cup \set{\set{0}}
\stopformula
```

$$s(1) = s(\{0\}) = \{0\} \cup \{\{0\}\} \quad [14]$$

[14] s of group 1 end group equals s of group the set 0 end the set end group equals the set 0 end the set union the set the set 0 end the set end the set

```
\startformula
  a\sqrt{x} = ax^{1/2} \neq ax^{1/3} = a\root[3]{x}
\stopformula
```

$$a\sqrt{x} = ax^{1/2} \neq ax^{1/3} = a\sqrt[3]{x} \quad [15]$$

[15] a times the square root of x equals a times x to the power of group 1 divided by 2 end group is not equal to a times x to the power of group 1 divided by 3 end group equals a times the root with degree 3 of x

```
\startformula
  \rationals = \set{\frac{p}{q} \fence p,q \in \integers \land q \neq 0}
\stopformula
```

$$\mathbb{Q} = \left\{ \frac{p}{q} \,\middle|\, p, q \in \mathbb{Z} \land q \neq 0 \right\} \quad [16]$$

[16] the rational numbers equals the set the fraction of p and q such that p comma q belongs to the integers and q is not equal to 0 end the set

```
\startformula
  f \mapsas x \mapsto x + \exp(x)
\stopformula
```

$$f : x \mapsto x + \exp(x) \quad [17]$$

[17] f is defined so that x maps to x plus exp of x

```
\startformula
  \lim_{k \tendsto +\infty} \frac{A__k}{B__k}
\stopformula
```

$$\lim_{k \to +\infty} \frac{A_k}{B_k} \quad [18]$$

[18] the limit under group k tends to plus infinity end group the fraction of numerator A with lower index k end numerator and denominator B with lower index k end denominator

```
\startformula
  \Gamma__1^^2__3^^4 \neq \Gamma__1^^2^^{}__3^^4
\stopformula
```

$$\Gamma_{1\,3}^{2\,4} \neq \Gamma_{1\,3}^{2\,4}$$ [19]

[19] Γ postscripts sub 1 super 2 sub 3 super 4 end scripts is not equal to Γ postscripts sub 1 super 2 sub 3 super 4 end scripts

```
\startformula
  \int_{a}^{b} f'(x) \dd x = f(b) - f(a)
\stopformula
```

$$\int_a^b f'(x)\,dx = f(b) - f(a)$$ [20]

[20] integral from a to b , of f prime of group x end group differential d x equals f times group b end group minus f times group a end group

```
\startformula
  \int_{\Omega} f \dd \mu = 0
\stopformula
```

$$\int_\Omega f\,d\mu = 0$$ [21]

[21] integral over Ω , of f differential d μ equals 0

```
\startformula
  \sigma \of (A \transpose{A}) \setminus \set{0}
  =
  \sigma \of (\transpose{A} A) \setminus \set{0}
\stopformula
```

$$\sigma(AA^T) \setminus \{0\} = \sigma(A^T A) \setminus \{0\}$$ [22]

[22] σ of group A times the transpose of A end group set minus the set 0 end the set equals σ of group the transpose of A times A end group set minus the set 0 end the set

```
\startformula
  \frac{\partial^3 u}{\partial x^2 \partial y}
\stopformula
```

$$\frac{\partial^3 u}{\partial x^2\,\partial y}$$ [23]

[23] the partial derivative partial d cubed u over partial d x squared partial d y end derivative

34.13 Development

Getting all these meanings right was sort of trivial with all the already present tools in ConTEXt. We will not go into details but it basically boils down to the following:

- A user keys in a formula, preferably using the proper, structure related commands, but this comes natural.
- A raw formula (noad list) enters a chain of Lua processing, this has always been the approach in MkIV, so nothing new here.
- Although there is some tagging happening at the TEX end, we also intercept some (yet) missing cases in Lua, again that already happened.

- When a formula is typeset, the result is turned into an MathML representation, and here we just use the existing export facilities of ConTEXt.
- The now xml encoded formula is then used for generating the meaning, which is the new thing in LMTX.

If tagging is also needed we embed the MathML as well as the meaning, so contrary to MkIV, we no longer tag the individual components but treat the formula as a whole. After all, this sequential tagging made little sense anyway.

When we go from MathML to meaning, we use information about characters that is part of a dictionary subsystem: we put symbols into categories which permits us to render with different spacing etc. based on the domain we're dealing with but it also permits assigning a different meaning. This math sub-project gave us a good reason to finally add this dictionary information to the character database (which of course gave us some good laughs about how weird math sometimes get coded, also in the perspective of Unicode).

Right from the start we decided that for real useful accessibility we needed to support multiple languages, which is why we started with simultaneous English and Swedish. We will add support for Dutch and depending on needs and input other languages. Such additions are relatively easy and fit well into the way ConTEXt is set up. In figure 34.1 and 34.2 you see a few pages from a large document with examples that we used for development. This document will be in the ConTEXt distribution and will be extended when there is need for more. It can also help translators. The document can show more details, like dictionary information, if needed.

$C((a,b)) \neq C^2([a,b]) \neq C^2[0,1] \neq C(\Omega) \neq \mathbb{C}(\Omega)$

C APPLYFUNCTIONOF BEGIN GROUP OPTIONAL BEGIN OPENINTERVAL a COMMA b END OPENINTERVAL END GROUP IS NOT EQUAL TO C SUPINDEX 2 APPLYFUNCTIONOF BEGIN GROUP OPTIONAL BEGIN interval a COMMA b END interval END GROUP IS NOT EQUAL TO C SUPINDEX 2 APPLYFUNCTIONOF OPTIONAL BEGIN interval 0 COMMA 1 END interval IS NOT EQUAL TO C APPLYFUNCTIONOF BEGIN GROUP Ω END GROUP IS NOT EQUAL TO C APPLYFUNCTIONOF BEGIN GROUP Ω END GROUP

C of group the open interval a comma b end the open interval end group is not equal to C with upper index 2 of group interval a comma b end interval end group is not equal to C with upper index 2 of interval 0 comma 1 end interval is not equal to C of group Ω end group is not equal to C of group Ω end group

C av grupp det öppna intervallet a komma b slut det öppna intervallet slut är inte lika med C med övre index 2 av grupp interval a komma b slut interval slut är inte lika med C med övre index 2 av interval 0 komma 1 slut interval är inte lika med C av grupp Ω slut är inte lika med C av grupp Ω slut

```
% C \of examples
% We shall not get rid of the grouping since it gives structure
% One could think of a \nogroup (just as \notimes)
\im { C \of (\openinterval{a,b}) \neq C^^2 \of (\interval{a,b}) \neq C^^2 \of \interval{0,1} \neq C\of(\Omega) \neq C \of (\Omega)}
```

```
<math>
 <mrow>
  <mi>C</mi>
  <mo></mo>
  <mo>(</mo>
  <mrow>
   <mo>(</mo>
   <mrow>
    <mi>a</mi>
    <mo>,</mo>
    <mi>b</mi>
   </mrow>
   <mo>)</mo>
  </mrow>
  <mo>)</mo>
  <mo>≠</mo>
  <msup>
   <mi>C</mi>
   <mn>2</mn>
  </msup>
  <mo></mo>
  <mo>(</mo>
  <mrow>
   <mo>[</mo>
   <mrow>
    <mn>0</mn>
    <mo>,</mo>
    <mn>1</mn>
   </mrow>
   <mo>]</mo>
  </mrow>
  <mo>≠</mo>
  <mi>C</mi>
  <mo></mo>
  <mo>(</mo>
  <mi>Ω</mi>
  <mo>)</mo>
  <mo>≠</mo>
  <mi>C</mi>
  <mo></mo>
  <mo>(</mo>
  <mi>Ω</mi>
  <mo>)</mo>
 </mrow>
</math>
```


Figure 34.1 The file with examples that we test.

Figure 34.2 The file with examples that we test.

34.14 A longer example, revisited

We show below again the example from the introduction, this time with the math interpretations written out. We show this on separate pages in because here we have a somewhat complex mixed tracing situation (here we use the Stix font).

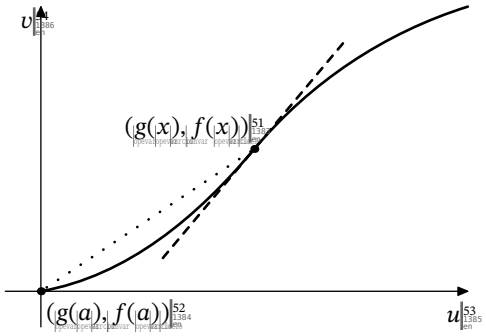


Figure 34.3

Proving the l'Hospital rule directly from the Lagrange mean value theorem

Anders Holst
Mikael P. Sundqvist

Abstract. At our first-year calculus course for engineers we discuss Lagrange's mean value theorem but not Cauchy's mean value theorem, and for this reason we usually give a weak form of l'Hospital's rule on limits. In this note we give a simple proof of the stronger version of l'Hospital's rule, using only Lagrange's mean value theorem and elementary results on limits and derivatives.

We formulate and prove the l'Hospitals rule for one-sided limits. This in fact strengthen the usual formulation slightly.

Theorem 34.1 (l'Hospital's rule)**.** *Assume that the functions f and g are continuous in $[a, b)$ and differentiable in (a, b). Assume further that $f(a) = g(a) = 0$ and that $g'(x) \neq 0$ in (a, b). If $f'(x)/g'(x) \to A$ as $x \to a^+$, then $f(x)/g(x) \to A$ as $x \to a^+$.*

A geometric interpretation of the l'Hospital rule goes as follow. In the uv-plane, draw the curve parametrized by $u = g(x)$ and $v = f(x)$. Then the direction coefficient $f(x)/g(x)$ of the secant (dotted in Figure 34.3) connecting $(g(x), f(x))$ with $(g(a), f(a)) = (0, 0)$ should approach the same value as the direction coefficient $f'(x)/g'(x)$ of the tangent to the curve at $(g(x), f(x))$ (dashed in Figure 34.3) as x approaches a. Our proof of the theorem uses that we can parametrize this curve locally around the origin as a function graph $u = t$ and $v = f(g^{-1}(t))$.

The only place in our proof where Lagrange's mean value theorem occurs is in this useful property of right-hand side derivatives.

Lemma 34.2. *Let $c > 0$. Assume that $\phi : [0, c) \to \mathbb{R}$ is continuous in $[0, c)$ and differentiable in $(0, c)$ and that $\lim_{t\to 0^+} \phi'(t)$ exists and equals A. Then*

$$\lim_{h\to 0^+} \frac{\phi(0+h) - \phi(0)}{h} = A.$$

Proof. For $h \in (0, c)$ the differential quotient $(\phi(0+h) - \phi(0))/h$ equals $\phi'(\xi_h)$ for some $\xi_h \in (0, h)$, by Lagrange's mean value theorem. As $h \to 0^+$ we have $\xi_h \to 0^+$ and so

$$\lim_{h\to 0^+} \frac{\phi(0+h) - \phi(0)}{h} = \lim_{h\to 0^+} \phi'(\xi_h) = A.$$

Proof (of Theorem 34.1)*.* Since g' is a Darboux function it will not change sign in (a, b) and for simplicity we assume that $g' > 0$ in this interval. Lagrange's mean value theorem assures that g is strictly monotone in the interval $[a, b)$ and thus that it has an inverse $g^{-1} : [0, g(b)) \to [a, b)$.

The composite function $\phi : t \mapsto f(g^{-1}(t))$, $t \in [0, g(b))$ is continuous at $t = 0$ and differentiable for $t \in (0, g(b))$. By the substitution $t = g(x)$ in the given limit, together with the chain rule and the rule of derivatives of inverse functions, we get

$$A = \lim_{x\to a^+} \frac{f'(x)}{g'(x)} = \lim_{t\to 0^+} \frac{f'(g^{-1}(t))}{g'(g^{-1}(t))} = \lim_{t\to 0^+} \frac{d}{dt} f(g^{-1}(t)) = \lim_{t\to 0^+} \phi'(t).$$

By Lemma 34.2, and by substitution $t = g(x)$ again, we conclude that

$$A = \lim_{t\to 0^+} \frac{\phi(0+t) - \phi(0)}{t} = \lim_{t\to 0^+} \frac{f(g^{-1}(t))}{t} = \lim_{x\to a^+} \frac{f(x)}{g(x)}.$$

This completes the proof.

35 Getting nostalgic

When Mikael and I were playing with Potrace we eventually ended up by turning some old rune font defined in MetaFont into a quite useable outline font. In the process we went from a Type3 to a proper OpenType (cff) instance. When doing this I remembered that in the times of active pdfTEX development, Thanh, Taco and I played with what we called pdf glyph containers. I had done experiments with using xforms (small images) as replacement for bitmap characters but that is not what we want with fonts, so after discussing it with Thanh Type3 support was extended with a user driven variant. Due to the way MetaFonts are defined the results were mixed. Filled shapes could be made to work but outlines often looked bad and much had to do with the fact that these fonts were never defined with odd-even rules in mind. So, Taco made a bitmap to outline program that we could hook in and in the end we had a reasonable workflow. We're talking of the period 1997 to 1998, some 25 years ago!

So, when in 2023 we came back to bitmap fonts in the perspective of LuaMetaTEX, some memories returned but I couldn't find the files or code at that time, which might relate to the fact that I always start from specifications. However, when in January 2024 I had to fix some issue with embedding an old Type1 font and after succeeding in that I decided again to search for that old code. This time I just grep'd the pdfTEX code base (as I had the TEXLive git checkout) and I found back this feature. In the end I had been looking at the wrong 'strings' because it looked like the interface evolved a little.

The basics are as follows. When we have a file (in pdfTEX) the tfm blob is what matters for the frontend. The backend that does the inclusion links that to a suitable font resource, most likely in Type1 format. This is specified in a map file that also permits specifying an encoding vector. For some reason it is not possible to map to other font formats other than Type1 and TrueType. No mapping means that a pk file is used and then we end up in some complicated semi automatic bitmap generation.

When I tested how these containers worked out nowadays, once pdfTEX[35] is happy to include a font it will bark about a missing font resource: it always wants a bitmap. Now, you can complain about this but I bet no one ever used this 25 year old feature so no one ever requested a check for a `pgc` file. Basically, when such a file is found instead of a `600pk` file, all should be fine, even if that is a zero length file. Given that the backend actually checks for it and not considers a container file useable, one can consider it an oversight, or maybe it just got lost in the kpse library that manages pk fonts and its generation.

Anyway, so how does it work? When the backend checks for a glyph container it expects this:

```
\pdffontscale factor
\pdfglyph slot width height depth llx lly urx ury =
    % pdf operators
\endglyph
```

One can add a `\pdffont` and `\endfont` wrapper because pdfTEX only searches for the mandate scale and one or more glyphs. A simple test then becomes:

- Copy a tfm file to a new name, say `cmr10mine.tfm`.
- Create a bogus pk file, for instance a zero byte `cmr10mine.600pk`.
- Make sure it can be found, I had to set `set MISCFONTS=.`.
- Create a glyph container with suffix `pgc` with some valid pdf code.

And then use a safe character for testing present in most fonts at the same position in the used encoding:

[35] Version 3.141592653-2.6-1.40.25 and older.

```
\starttext
    \font\test=cmr10mine \test \char 65 % A
\stoptext
```

This should produce something, assuming an entry like:

```
\pdffontscale 0.01
\pdfglyph 65 200 0 0 0 0 200 200 =
    10 M 1 j 1 J 5 w
    % outlines
    h S
\endglyph
```

Just for completeness, I could find a Perl program on my disk that I wrote at that time and that has the banner:

```
mptopgc 1.01 - pdfTeX / HH 1998
```

It produces help like:

```
       --silent   use logfile instead of screen (no)
      --autorun   run MetaPost first (no)
     --optimize   convert gray into eofill (no)
       --ignore   ignore gray path (no)
         --gray   permit the gray operator (no)
        --merge   substitute gray using glyphs (no)
--magnification   set MetaPost mag factor (1)
     --rounding   alternative rounding offset (0)
      --pgcpath   pgc path
      --tfmpath   tfm file path
      --tfmtopl   ftmtopl binary name (tfmtopl)
     --metapost   metapost binary name (mpost)
```

But it is a long time since I installed Perl on my laptop so I didn't check it. I also found a manual that could be updated but I don't see much use for it now. There is even a MkII TEX file that one can run with `pdftex &plain \input supp-pgc.mkii \MPtoPGC` that produces container files but I didn't test that either.

However, what I did test was if it still works. For that I used the following test file:

```
% engine=pdftex

\startTEXpage[offset=10pt]

\pdfmapline{texnansi-lmr10 < texnansi.enc}

\font\test=texnansi-lmr10 \test

\input tufte

\stopTEXpage
```

Here I use a normal tfm file because after all we need the metrics. Then I made an empty `texnansi-lmr10.600pk` so that we can fool the pk generator. Using Latin Modern files is more convenient. The glyph container file was made by abusing some features of LMTX:

- With the module `s-fonts-outlines` I made a font sample that as byproduct generates a file with outline definitions. Basically any attempt to use an outline in MetaFun will populate the cache. We load the font `lmr10.afm` and it will get the outlines from `lmr10.pfb`. Of course there are more that 256 characters in there but that doesn't matter.

- I also generated a `texnansi` vector file which can be done from a normal TEX run using the Lua call `fonts.encodings.load ("texnansi.enc"))`. It locates and converts a regular `enc` into a Lua representation. I picked up the file from the cache.

- From these files I generated a glyph container file `texnansi-lmr10.pgc` that can be used instead of the pk file. The file `mtx-pgc.lua` is shown below.

- I made sure to set `MISCFONTS` to a sensible value because otherwise the container file is not found (who knows what location strategy is used).

This is the 'quick and dirty' conversion file:

```
local round = math.round
local concat = table.concat

local result = { "\\pdffontscale 0.001\n" }
local r      = 1

local f_pdfglyph = string.formatters["\\pdfglyph %i %i %i %i %i %i %i %i"]
local s_endglyph = "\\endglyph\n"
local f_comment  = string.formatters["%% name %s index %i slot %i \n"]

local shapes   = table.load("lmr10-pfb.tma")
local encoding = table.load("texnansi.tma")

local glyphs = shapes.glyphs
local hash   = encoding.hash

for index=0,#glyphs do
    local glyph    = glyphs[index]
    local segments = glyph.segments    or { }
    local bounds   = glyph.boundingbox or { 0, 0, 0, 0 }
    local width    = round(glyph.width or 0)
    local name     = glyph.name
    local slotused = hash[name]
    if slotused then
        local llx, lly = bounds[1], bounds[2]
        local urx, ury = bounds[3], bounds[4]
        r = r + 1 ; result[r] = f_comment(name,index,slotused)
        r = r + 1 ; result[r] = f_pdfglyph(slotused,width,0,0,llx,lly,urx,ury)
        if #segments > 0 then
            r = r + 1 ; result[r] = "10 M 1 j 1 J 0 w"
            for i=1,#segments do
                local s = segments[i]
                if s[1] ~= "close" then
                    r = r + 1 ; result[r] = concat(s," ")
```

```
                end
            end
            r = r + 1 ; result[r] = "h f*"
        end
        r = r + 1 ; result[r] = s_endglyph
    end
end

io.savedata("texnansi-lmr10.pgc",concat(result,"\n"))
```

Of course for a proper workflow we need to make this into a real helper script but here I just want to show the idea. The result is shown in figure 35.1. There is however one drawback: because pdfTEX goes via the pk route using the same font with a different scale will result in an extra copy while actually we have a scalable Type3 font. It is for that reason that I will not waste any more time on this. I could provide a patch to deal with it but changing a long time ago frozen TEX program is tricky because bug(let)s have become features.

> We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 35.1 This is what we get with glyph containers.

So where does this bring us? Because MkII is frozen and pdfTEX only works with that version it makes no sense to add some kind of Type3 support. In MkIV and MkXL we have several mechanisms for creating these fonts anyway. I cannot comment on use outside ConTEXt but because of the strong connection with pk fonts being required in order for glyph containers to kick in at all, I suspect that it's a mostly unknown feature, but one that actually can be made to work. It could even work better (with respect to file lookup and font scaling) when it is decoupled from the pk requirements and listen to na map file line that also signals if the font can be reused. But what worries me a bit more is that after 25 years I hardly recognize the (reasonable) Perl code and little manual although it slowly came back. At that I time I could not have predicted where we are now with respect to Type3 support, which in LMTX is very much kicking and alive.[36]

[36] At that time Type3 in the perspective of TEX always had a bad ring, probably because the many files that were around with too low resolution bitmap fonts. The pdf readers were not particularly good in rendering them either. But with outlines this format can be fun!

36 PDF 2.0

The pdf file format has evolved over decades and support in TEX macro packages has evolved with it. The pdfTEX engine defaults to version 1.4 but ConTEXt MkII bumps that to 1.5 by default. The LuaTEX engine initializes to 1.0 but in ConTEXt MkIV we use 1.7 by default, although one can choose some standard that uses a different value. A quick check shows that XETEX, that uses dvipdfmx goes for 1.5 by default. In LuaMetaTEX we default to nothing because it has no backend built in, but in MkXL we also use 1.7 as default. So, where does the latest greatest pdf 2.0 fit in?

It's good to notice that the difference between version 1.7 and 2.0 is not that large, especially when we look at a TEX engine. When we talk about text we only have to provide page streams with text rendering operators and embed fonts that make this possible. We can support color by pushing the right operators and, if needed, graphic state objects in the file. While TEX only has rules, we're fine with simple drawing operations. We can use other drawing operators when we convert from MetaPost or use packages like TikZ. Of course adding hyperlinks and alike is possible and here we can try to play safe, but we can also introduce issues because we rely on the viewer. If we insert graphics we have to make sure that the right objects are injected, and here we are dependent of the quality of the producer of those graphics. In most aspects an upgrade to 2.0 is no big deal if we already can provide 1.7 output.

Because pdf is used for printing, interchange and archiving, some standards have evolved that put restrictions on what can go in the pdf file. If we decide to go for 2.0 output, we can forget about these standards because pdf 2.0 supports the lot. It is somewhat more restricted and features that are deprecates sometimes are in fact obsolete and supposed not to be used. This is puzzling because there is no real need to drop something that is already supported. The main problem here is that validators can be picky. There are also amendments made to 2.0, some of which smell like accommodating applications that have issues with them. It's not like we have tons of old viewers that are used large scale that are not up to 2.0 quality pdf. Maybe we should just forget about the pre 2.0 standard profiles.

Tagging has been part of pdf for a while but in the beginning was mostly related to applications marking content in a way useful for those applications. When embedding a page from a file that information is rather useless. Later accessibility became an issue and that kind of tagging was added and later adapted a bit in pdf 2.0, but in 2024 it is still pretty much unclear how to deal with it, if only because in 15 years no real development has taken place in viewers when it comes to using these features. The best we can do is to make sure that validators are happy with what ConTEXt produces.

It must be noted that when we talk accessibility, the fact that embedding audio and video went from being easy to being complicated, with an intermediate flirt with flash (in it self okay SMIL). In a similar way interactive features are a bit unstable, and for instance simple (trivial) viewer control would have made live much easier. In order to really be accessible one should just ship dedicated versions of documents, its source or, if one considers that more usable, some kind of html output (of course that's often a short term perspective).

So, what is needed to support 2.0 in ConTEXt? The output that we render is basically the same. We can leave out some details like so called cidsets and procsets and we have to make sure that some demands are met, like mandate resources and using registered tags for private entries in dictionaries. All that is easy compared to inclusion of third party content: here we often need to do some cleanup in order to pass the tests. For that we need not only check some specific dictionaries but also parse the page (and form) content streams and if needed clean them up. This comes a (runtime) price but it's bearable. More tricky is dealing with missing (or bad) fonts but again it can be done, maybe with a dedicates configuration to control the process. See the `pdfmerge` manual for more information about this.

When it comes to tagging the best we can so is play safe and rely on future content analysis helped with generic tags. After all that us what these language models promise us. It doesn't hurt to be somewhat sceptic with regards to standards and the future. Just look at how typesetting, printing and publishing evolved. Technology is not that long term stable and with big tech in charge commerce drives most of it. It's already a miracle is some application or technology survives a few years. So, we really need to restrict ourselves to what makes sense.

All that said: we're ready to deal with 2.0 and can always adapt if needed. For that purpose we also embed additional ConTEXt specific information so that in the future we can, if needed, upgrade a pdf file, although with a TEX tool chain one can always regenerate a document. The main question is, when do we default to it.