MAPS

NUMMER 52 • VOORJAAR 2022

NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

Voorzitter
Hans Hagen
voorzitter@ntg.nl

Secretaris
Taco Hoekwater
secretaris@ntg.nl

Penningmeester
Robbert Schwippert
penningmeester@ntg.nl

Bestuursleden
Frans Goddijn
Pieter van Oostrum

Postadres
Nederlandstalige TEX Gebruikersgroep
Baarsjesweg 268-I
1058 AD Amsterdam

ING bankrekening
IBAN: NL53INGB0001306238
BIC: INGBNL2A

E-mail bestuur
ntg@ntg.nl

E-mail MAPS redactie
maps@ntg.nl

WWW
www.ntg.nl

De **Nederlandstalige TEX Gebruikersgroep (NTG)** is een vereniging die tot doel heeft de kennis en het gebruik van TEX te bevorderen. De NTG fungeert als een forum voor nieuwe ontwikkelingen met betrekking tot computergebaseerde documentopmaak in het algemeen en de ontwikkeling van 'TEX and friends' in het bijzonder. De doelstellingen probeert de NTG te realiseren door onder meer het uitwisselen van informatie, het organiseren van conferenties en symposia met betrekking tot TEX en daarmee verwante programmatuur.
De NTG biedt haar leden ondermeer:

- Tweemaal per jaar een NTG-bijeenkomst.
- Het NTG-tijdschrift MAPS.
- De 'TEX Live'-distributie op DVD/CDROM inclusief de complete CTAN software-archieven.
- Verschillende discussielijsten (mailing lists) over TEX-gerelateerde onderwerpen, zowel voor beginners als gevorderden, algemeen en specialistisch.
- De FTP server `ftp.ntg.nl` waarop vele honderden megabytes aan algemeen te gebruiken 'TEX-producten' staan.
- De WWW server `www.ntg.nl` waarop algemene informatie staat over de NTG, bijeenkomsten, publicaties en links naar andere TEX sites.
- Korting op (buitenlandse) TEX-conferenties en -cursussen en op het lidmaatschap van andere TEX-gebruikersgroepen.

Lid worden kan door overmaking van de verschuldigde contributie naar de NTG-giro (zie links); vermeld IBAN zowel als SWIFT/BIC en selecteer shared cost. Daarnaast dient via `www.ntg.nl` een informatieformulier te worden ingevuld. Zonodig kan ook een papieren formulier bij het secretariaat worden opgevraagd.
De contributie bedraagt € 35. Voor studenten geldt een tarief van € 18. Dit geeft alle lidmaatschapsvoordelen maar *geen stemrecht.* Een bewijs van inschrijving is vereist. Een gecombineerd NTG/TUG-lidmaatschap levert een korting van 10% op beide contributies op. De prijs in euro's wordt bepaald door de dollarkoers aan het begin van het jaar. De ongekorte TUG-contributie is momenteel $105.

Afmelding kan met ingang van het volgende kalenderjaar door opzegging per e-mail aan de penningmeester.

MAPS bijdragen kunt u opsturen naar `maps@ntg.nl`, bij voorkeur in LATEX- of ConTEXt formaat. Bijdragen op alle niveaus van expertise zijn welkom.

Productie. De Maps wordt gezet met behulp van een LATEX class file en een ConTEXt module. Het pdf bestand voor de drukker wordt aangemaakt met behulp van pdftex 1.40.20 en luatex 1.13.0 draaiend onder MacOS X 12.2. De gebruikte fonts zijn Linux Libertine, het niet-proportionele font Inconsolata, schreefloze fonts uit de Latin Modern collectie, en de Euler wiskunde fonts, alle vrij beschikbaar.

TEX is een door professor Donald E. Knuth ontwikkelde 'opmaaktaal' voor het letterzetten van documenten, een documentopmaaksysteem. Met TEX is het mogelijk om kwalitatief hoogstaand drukwerk te vervaardigen. Het is eveneens zeer geschikt voor formules in mathematische teksten.
Er is een aantal op TEX gebaseerde producten, waarmee ook de logische structuur van een document beschreven kan worden, met behoud van de letterzet-mogelijkheden van TEX. Voorbeelden zijn LATEX van Leslie Lamport, $\mathcal{AMS}$-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Contents

Welcome

Abstract
Door middel van de Maps willen we u op de hoogte houden van ontwikkelingen, ook om daarmee onze leden te danken voor hun trouwe steun aan de TEX ontwikkelaars. Verder bieden we ruimte aan lezers die anderen laten delen in hun ervaringen met TEX, MetaPost, fonts en aanverwanten. Aarzel dus niet ons artikelen te sturen. Een halve pagina is al heel leuk, meer mag ook, graag zelfs. Het hoeft geen ‚zware kost' te zijn want het is voor lezers bijvoorbeeld al heel interessant te lezen hoe anderen TEX gebruiken. Dus een artikeltje als „dit doe ik met TEX, zo doe ik dat en nu kun jij het ook" is zeer welkom!
Hoewel het internet tegenwoordig een belangrijke bron van informatie is, blijft papier een functie vervullen binnen de vereniging. Dat past immers bij TEX!

The TEX ecosystem has evolved in a typesetting environment that can be used for a wide variety of documents. But, in this maps there are some articles that, as can be expected given TEX's objectives, discuss rendering math.

In the Lincos book, the Dutch mathematician Hans Freudenthal describes a language (system) that can be used to communicate with aliens. There is a strong focus on communicating math. Just browsing the book is fun already. Now, imagine that you have to typeset this or an article based on it. Say that Hans writes it, then Taco has to make sure it gets rendered properly in the maps style. After that Frans starts proofreading. All three need to check proper spacing of the formulas. That can become a tedious cycle.

But . . . this document was published in the 1960s when there was no TEX! How much easier life has become. On the next page we show an example: page 98, definition 3 01 8, about commenting. More on this book can be found at

```
https://en.wikipedia.org/wiki/Lincos_language
```

and a scan of this book can be downloaded from:

```
https://monoskop.org/images/8/85/Freudenthal_Hans_Lincos_Design
              _of_a_Language_for_Cosmic_Intercourse_Part_I.pdf
```

Given the care that Don Knuth paid to his books and TEX being able to do well there is very little excuse for authors that embed math in documents today to produce sloppy output. Nevertheless we can easily run into badly rendered math on the web today. However, it is all about paying attention and Freudenthal and Knuth both show us the way. Hopefully this Maps will pass your quality criteria.

* $^{t_5}Ha\,\mathrm{Inq}\,Hb\,.\,t_1\,t_2\,\mathrm{Fit}\,\mathfrak{p}^{t_4}$: $Hb\,\mathrm{Inq}\,Hc\cdot t_3\,t_4\,Ha\,\mathrm{Inq}\,Hb\,.\,t_1\,t_2\,\mathrm{Fit}\,\mathfrak{p}^{t_5}$:
$Hc\,\mathrm{Inq}\,Hd\cdot t_4\,t_5\,Hb\,\mathrm{Inq}\,Hc : t_3\,t_4\,Ha\,\mathrm{Inq}\,Hb\,.\,t_1\,t_2\,\mathrm{Fit}\,\mathfrak{p}$ *

are better suited for this purpose. We shall send a great many texts from which this will become still more evident.

3 01 8. Comments on some texts may be very useful. E.g. on the first talk of 3 01 2:

* $Hc\,\mathrm{Inq}\,Hd$: $t_1\,t_2\,Hb\,\mathrm{Inq}\,Ha\cdot 10\,x = 101\,.\rightarrow .\,x = 101/10$ ·
$Hd\,\mathrm{Inq}\,Hc\,\mathrm{Ben}$ *

A comment on the second talk of 3 01 2:

* $Hc\,\mathrm{Inq}\,Hd$ ·
$t_1\,t_2\,Ha\,\mathrm{Inq}\,Hb\cdot ?\,x\,.\,10\,x = 101 : \wedge : t_2\,t_3\,Hb\,\mathrm{Inq}\,Ha\,.\,101/10$:
$Hd\,\mathrm{Inq}\,Hc\,\mathrm{Ben}$ *

Another comment on the second talk of 3 01 2:

* $Hc\,\mathrm{Inq}\,Hd\cdot t_3\,t_4\,Ha\,\mathrm{Inq}\,Hb : {}^{\curlyvee}x\,.\,t_2\,t_3\,\mathrm{Fit}\,x\cdot \in \mathrm{Ben}$:
$Hd\,\mathrm{Inq}\,Hc\,\mathrm{Ben}$ *

A comment on the third talk of 3 01 2:

* $Hc\,\mathrm{Inq}\,Hd\cdot t_2\,t_3\,Ha\,\mathrm{Inq}\,Hb : Hb\,\mathrm{Inq}\,Ha\,101/10\,.\in \mathrm{Ben}$:
$Hd\,\mathrm{Inq}\,Hc$: Fal : $t_2\,t_3\,Ha\,\mathrm{Inq}\,Hb : {}^{\curlyvee}x\,.\,t_1\,t_2\,\mathrm{Fit}\,x\cdot \in \mathrm{Ben}$:
$\neg$: $t_2\,t_3\,Ha\,\mathrm{Inq}\,Hb : Hb\,\mathrm{Inq}\,Ha\,101/10\,.\in \mathrm{Ben}$ ·
$\leftrightarrow$ · $t_2\,t_3\,Ha\,\mathrm{Inq}\,Hb : {}^{\curlyvee}x\,.\,t_1\,t_2\,\mathrm{Fit}\,x\cdot \in \mathrm{Ben}$ *

From these comments the receiver will learn what liberties a person may take when quoting other people. One could add a hypercomment put into the mouth of still other persons and containing behaviour rules on honest quoting.

In a former version of Lincos we distinguished between literal and free quotations by means of a special notation which was dropped later on. Literal quotation is a rather unimportant limit position. We shall develop a means of comparing the exactness of quotations ('Err', 3 09 1, 3 19 1). This will prove to be more useful. If needed, literal quotations may be characterized by 'Err . . . = 0'.

3 02 0. We shall here treat interrogative sentences:

3 02 1. Many interrogative pronouns and adverbs can be treated in the following manner:

* $^{t_1}Ha\,\mathrm{Inq}\,Hb\cdot ?\,x\,.\,100\,x = 1010^{t_2}$:
$Hb\,\mathrm{Inq}\,Hc\cdot ?\,y : t_1\,t_2\,y\,\mathrm{Inq}\,Hb\cdot ?\,x\,.\,100\,x = 1010$:
$Hc\,\mathrm{Inq}\,Hb\,Ha$ *

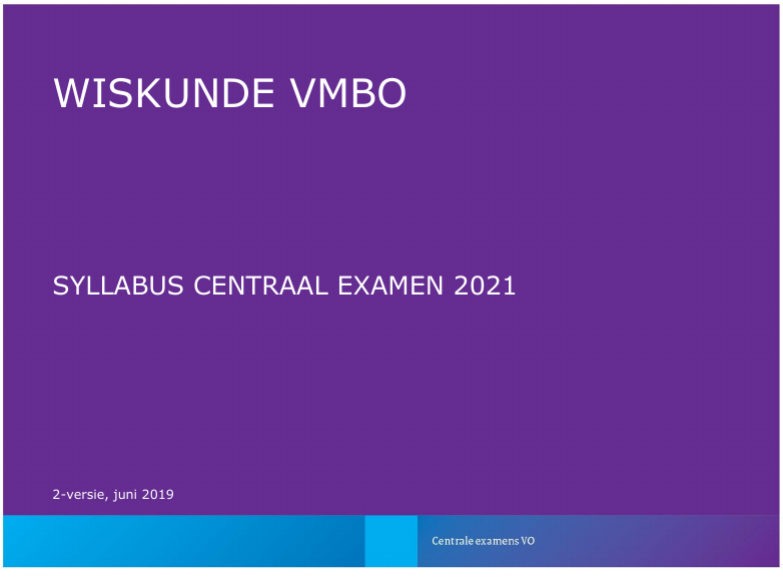
Uw redactie

Dutch Government Math rendering

End 2021 and beginning 2022 Mikael Sundqvist and I spent quite some time on an upgrade of the math engine. Because TEX itself is frozen that was done in LuaMetaTEX, which is our follow up on LuaTEX. That effort was all about consistency, avoiding side effects, optimized spacing and line breaks, compensating for issues in OpenType math fonts, interfaces and more. The TEX engine already does a good job on math, if only because it's one of the reasons for its existence and when looking at the way it's done one always needs to keep in mind the limitations of those days: memory, performance, font technology, etc. But with the arrival of OpenType math and after many years of working with TEX we took the opportunity to discuss and improve math typesetting in ConTEXt using new features of LuaMetaTEX.

When one spends so much time on something that is sort of a niche application (math) a valid question is “Who will benefit from it?”. Decades of observing TEX usage has made clear that it's mostly for users who like to make their document look nice. I'm not sure if publishers still care, as they outsource composition and often demand usage of word processors or visual markup tools. Even academic usage in for instance reports, course materials and thesis is questionable because not every TEX user cares for an non-voluntary usage of some program just for the sake of getting something on paper.

So, in the end all that effort on an upgraded engine is for the happy few who love to see things done right. Because ConTEXt has some focus on educational usage it is no surprise that I occasionally run into a document that targets education and also has some math. In this case I will show some usage of math in a document that describes what school kids have to learn. We're talking vocational education so one can imagine that a lot of attention is paid to lowering the boundaries, easing understanding, and being consistent in presenting the learning objectives. Alas.



This 34 page document is published by the Dutch government as can be derived from the logo shown next. And, as is usual today in documents, a reference to the website is there too, and (also as usual) it's likely some bitmap clip of some web page. We will see that the two lions in the banner don't represent the TEX lion here.

Now that we made clear that we're dealing with an official document, we can have a closer look at the math. From the examples here it will be clear that programs like TEX are wasted here and that such documents should never be used to determine the specifications of a typesetting program. There was a time that such a document would be in a typewriter font with handwritten formulas and I bet that it would look way better than what we will see here. It might even be easier to produce because I don't want to imagine what effort it takes to get this crappy output. The snippets shown are just selections from the document exported in png format. We keep the order of occurrence and scale a bit so that we can see clearly what we have.

$$\textit{totale prijs} = \textit{aantal personen} \times \textit{prijs}_{\textit{koffie}} + \textit{aantal personen} \times \textit{prijs}_{\textit{gebak}}$$
$$\textit{totale prijs} = \textit{aantal personen} \times (\textit{prijs}_{\textit{koffie}} + \textit{prijs}_{\textbf{\textit{gebak}}})$$

In this example we see a verbose example of a formula, which it itself is quite okay. It is however puzzling why in the second line the subscript text is in bold. The first line has some curious spacing, so it is definitely not done in so called math mode (assuming that the used word processor has that concept at all). A sans serif font is used so given that this is an example for school kids, one can wonder if the times ($\times$) is experienced different from an x (x), and we will see x's being used later. For the record, a math font setup has many variants: $\mathrm{x}+\mathbf{x}+x+\boldsymbol{x}+\mathsf{x}+\mathbf{x}+x+\boldsymbol{x}+\mathtt{x}+\mathbf{x}+\mathrm{x}+\mathrm{x} = \cdots$.

We stick a bit with verbose math and give another example. Let us be tolerant with respect to the interline spacing and just look at the math: we suddenly get a serif font here but definitely with a weight that does not really match the sans, so it is unlikely that a proper OpenType math font, with matching alphabets, has been used here so let's from now on assume that those responsible for rendering are unaware of the existence of such fonts. We can always blame the application.

$$\textit{hoeveelheid bloed} = \frac{1}{13} \times \textit{gewicht}\ (\textit{hoeveelheid bloed}\ \text{in liters en}\ \textit{gewicht}\ \text{in kg})$$

The next snippet shows an 'x' and as long as it is in an italic shape it will be different from the times symbol, but unfortunately in this document we see an interesting mix of upright, italic, sans and serif characters being used. At this point we probably no longer need to wonder why students in that segment of education have problems with math and why it is not that popular.

$$\text{verticale lijn } x = \mathrm{a}$$

An example of a different 'x' comes next. Also watch the somewhat out of proportion radical compared to the 'x'. In TEX one really has to bend rules to get that. One cannot select the root symbol so it might as well be some overlayed bitmap.

$$y = \sqrt{x}$$

We stay with the ‘x’ and get a slightly different italic serif this time. It is combined with a sans serif ‘y’ and upright somewhat bold ‘a’. Again, for this to be done in TEX one needs to exercise some effort because normally all come out in a math italic font. Spacing fractions is not always trivial especially when you see different ones alongside but consistency is nevertheless important.

$$y = \frac{\mathrm{a}}{x}$$

That the ‘x’ brings some artistic freedom is clear, but at least we have two similar shapes here. I'm not sure how one can explain to a student that this time we use an upright sans. It's probably all about the ‘x’ being smaller and raised.

$$y^{\mathrm{x}}$$

It is possible to have all symbols in italic, as is seen from the next snippet. Spacing could be a bit better but at least there is some consistency here.

$$h = 2t - 9$$

However, when one reads on this shows up:

$$y = \frac{1500}{x}$$

We do have two (this time serif) italic ordinary characters ‘x’ and ‘y’ but the number ‘1500’ made of digits is somewhat large and probably is typeset as an independent quantity.

As a welcome distraction we now show a table. The alignment in the first column is peculiar. As with much in this document it looks like there has been no proofreading at all. The numbers in the other cells are not (right) aligned and sit high in the cells and of course frames around the cells are used. A student might wonder if there is a difference between three and five dots. No mater how one abuses TEX, the commands that produce dots always produce the same amount: it's a proper glyph (shape)!

Afstand (in km)	1	2	5	10	20	...
Prijs (in euro’s)	9,10	10,20	13,50	19	.....	.....

Sometimes two successive lines indicate some concept (I guess) but that is no reason for this rendering. I cannot imagine that students are supposed to interpret such formulas depending on the inconsistent mix of fonts and weights being used. (In the following case using opp could have saved some space.)

$$\begin{aligned} \mathit{inhoud}_{\boldsymbol{\mathit{kegel}}} &= \frac{1}{3} \times \mathit{opp\ grondvlak} \times \mathit{hoogte} \\ \mathit{inhoud}_{\mathit{kegel}} &= \frac{\mathit{oppervlakte\ grondvlak} \times \mathit{hoogte}}{3} \end{aligned}$$

Here comes another beauty, a mix of digit '0' (or is it the letter 'O') and a degrees '°' symbol (I assume). In a decent (OpenType) math font the script symbols can have a shape optimized for the smaller size so one can't know I guess.

$$(0\ ^{o})$$

It is time again for a larger blob of text. Do you recognize the symbol pi (π) (it's not an 'n' but it comes close)? And what about the superscript digits '2' and '3' that also get special spacing? Weren't the digits upright in previous examples? The fractions look like some small image squeezed in with a non proportional scale.

$$\begin{aligned}
&\textit{omtrek cirkel} = \pi \times \textit{diameter}\\
&\textit{oppervlakte cirkel} = \pi \times \textit{straal}^{2}\\
&\textit{inhoud prisma} = \textit{oppervlakte grondvlak} \times \textit{hoogte}\\
&\textit{inhoud cilinder} = \textit{oppervlakte grondvlak} \times \textit{hoogte}\\
&\textit{inhoud kegel} = \frac{1}{3} \times \textit{oppervlakte grondvlak} \times \textit{hoogte}\\
&\textit{inhoud piramide} = \frac{1}{3} \times \textit{oppervlakte grondvlak} \times \textit{hoogte}\\
&\textit{inhoud bol} = \frac{4}{3} \times \pi \times \textit{straal}^{3}
\end{aligned}$$

By now you get the picture, so we show a few more in one go:

$$y = ax + b \qquad \boldsymbol{y = b\,g^{t}} \qquad \boldsymbol{y = a\,x^{n} + b}$$

In TEX there is a concept of a math axis, but not in the next example, and again one can wonder if the 'a' and 'x' come from the same font. I did not bother to disassemble the pdf. In TEX you can mess up the spacing too, but I get the impression that the 'x' is way below what a strut would enforce.

$$y = \frac{a}{x}$$

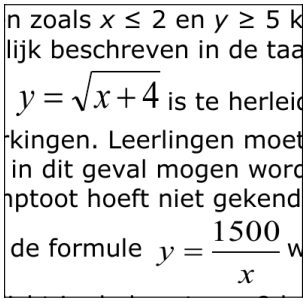
We started with a radical symbol that was somewhat high relative to what went under it. But it can't be worse than this. Not only do we have a squeezed radical symbol, the whole assembly also moved below the baseline: that takes some effort.

$$y = \sqrt{x}$$

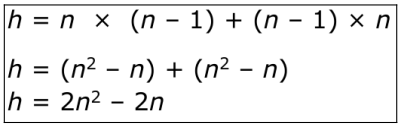
It is not uncommon to see some upright words in math formulas, think of sine and cosine operators, often used in conjunction with parentheses $\sin(x)$. But the inverse operator in the next example is special: it is not only in bold, but also negated. And it seems not to be an issue to show it combined with an upright bold 'y' raised to the power bold 'y' in spite of a previous also upright regular variant. If in this case calculator operations are meant, a more appropriate font or symbol should have been used.

$$y^{x} \text{ en } \mathbf{INV\text{-}\ y^{x}}$$

At some point one gets accustomed to this kind of rendering and maybe when that happens those who are supposed to (proof) read this will not notice anything weird and inconsistent in the larger clip below:



Not only are all ‘x’ and ‘y’ letters different, so are the digits and equal signs. It is hard to imagine that the thousands of readers of a document like this who have an education in math don't find this amusing. Personally it makes me sad. How can we expect students to pay attention to anything they have to produce if this kind of crap comes from the government. There was a time when such official documents were typeset by the state publisher that employed famous typesetters and proper printers. Even when much got delegated to departments responsible for communication that used typewriters with these math specific symbol bulbs there was professional pride involved. One can only wonder about the quality criteria that get applied today. There is simply no excuse for this and also not for the interline and inter atom spacing in the next one:



But maybe spacing has some meaning that I don't grasp because I cannot imagine that proof reading did not catch the next snippet, one that also uses very thin underlining:

Maybe documents like this don't get proofread. In fact, maybe they are not even (supposed to be) read. Maybe it's just some outsourced effort that ends up on a website and leaves the actual content to the teachers. Maybe one only has to look at some exams and drill and practice for what is in there. Maybe no one really cares.

rijnummer	1	2	3	4	..... .	10
aantal zitplaatsen	18	21	24	27	..... .	45

The table above actually starts the document and again proves that no one checked it, because I cannot imagine anyone not noticing the line breaks in the pre-last cells where the (this time) eight dots would have fit quite well.

So, what conclusion can we draw from this? First of all that there is a total lack of attention to how something looks and feels and thereby is perceived. Personally I am not willing to even consider this a serious document at all. If textual consistency is lacking then for sure the content is also not consistent and checked. And I did not even discuss the text, punctuation, spacing, usage of quotes and excessive use of frames around pages. We can only hope that documents like these get lost over time so that no one can wonder how badly typesetting has evolved since the middle ages.

It also reveals to me that working on TEX is really dedicated to users who do care and not to this kind of institutionalized math usage. But above all, it makes me aware of the fact that it is no wonder that math is unpopular among kids. If it looks like crap, it must be crap. We really should make math look 'cool' and 'super' these days and only using these buzz words when talking to kids is not enough! The good news is that after many decades TEX users can still produce nicely looking documents with plenty of math.

Like: $y = \sqrt{x+4}$ and $y = \frac{a}{x}$ and y^x and $h = 2t - 9$ and $y = \frac{1500}{x}$ and $y = ax^n + b$ and $\mathrm{inv}(-y^x)$ and $0°$ as well as: $\mathrm{inhoud}_{\mathrm{kegel}} = \frac{1}{3} \times \mathrm{oppervlakte\ grondvlak} \times \mathrm{hoogte}$, or: $\mathit{inhoud}_{\mathit{kegel}} = \frac{\mathit{oppervlakte\ grondvlak} \times \mathit{hoogte}}{3}$.

And you can mix in some colors, emoji, graphics and still be consistent. If you don't pay attention to your readers, don't expect your readers to pay attention to what you bring to the table. And, once you know how to use TEX it's pretty easy and even saves time, because even getting a handful of formulas as bad as seen here takes time.

If you still wonder why we should care about these matters, imagine that you need new tires for a car and get it back with four differently sized ones. How would driving that car feel to you and would you be willing to keep that configuration for the time it takes to wear them off? Given that math (and teaching it) is pretty much about consistency, I suppose that when the rendering of math as shown here doesn't disturb you, you will also happily keep those different tires.

Hans Hagen

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.[1] 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

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

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 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:

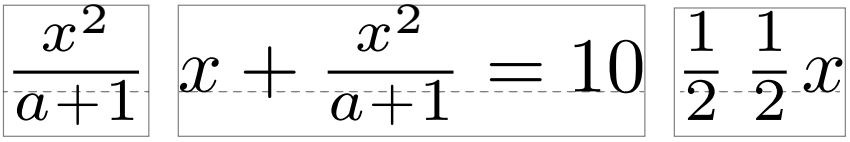
```
\ruledhbox{$\frac{x^2}{a+1}$}
```

and:

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

as well as:

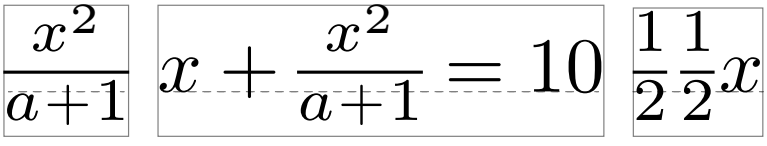
```
\ruledhbox{$\frac{1}{2}\frac{1}{2}x$}
```



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, XƎTEX 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:



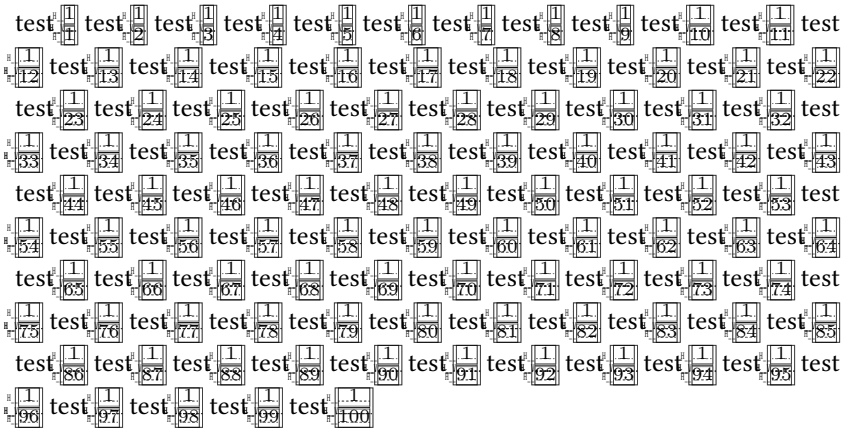
As intermezzo and moment of contemplation I show some examples of fractions mixed into text. When we have the delimiter space set we get this:

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 $\frac{1}{100}$

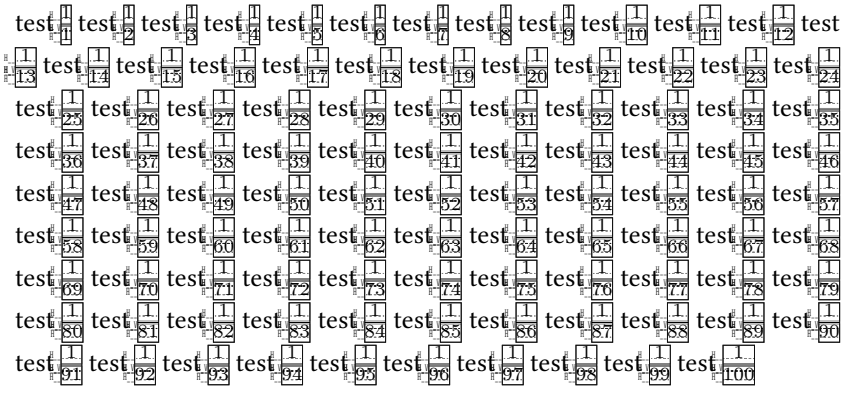
While with zero it looks like this, quite a different outcome:

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:

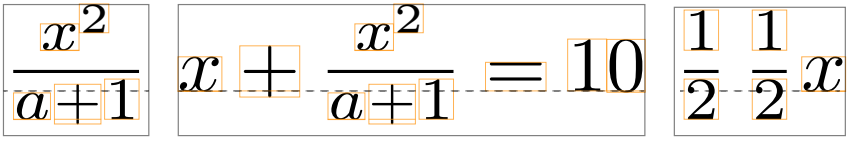


You can zoom in and see where it interferes with margin alignment.

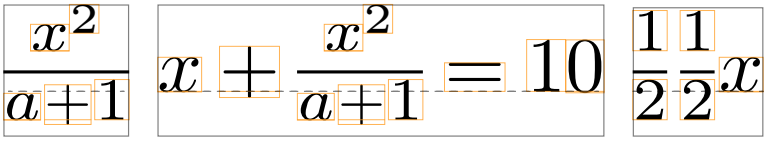


So, if you ever meet a user who claims perfection and superiority of typesetting, check out her/his work which might have inline fractions done the spacy way. It might make other visually typesetting claims less trustworthy. And yes, one can wonder if margin kerning could help here but as this content 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):

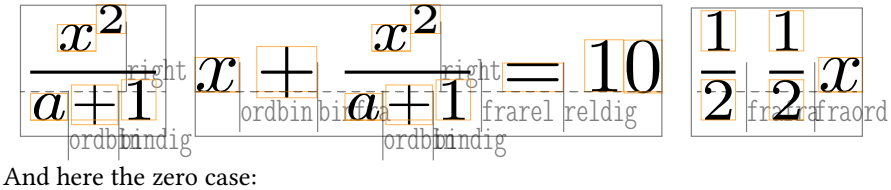


Again we also show the zero case

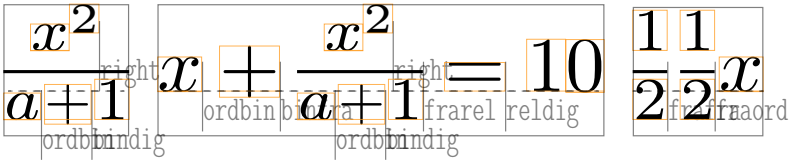


This makes clear why there actually is this extra space around a fraction: regular operators have side bearings and thereby have some added space. And when we put a fraction in front of a symbol we need that little extra space. Of course a proper class pair spacing value could do the job but there is 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:



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:[2]

```
\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
```

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

```
\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
```

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 $64 \times 64 \times 8$ 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.

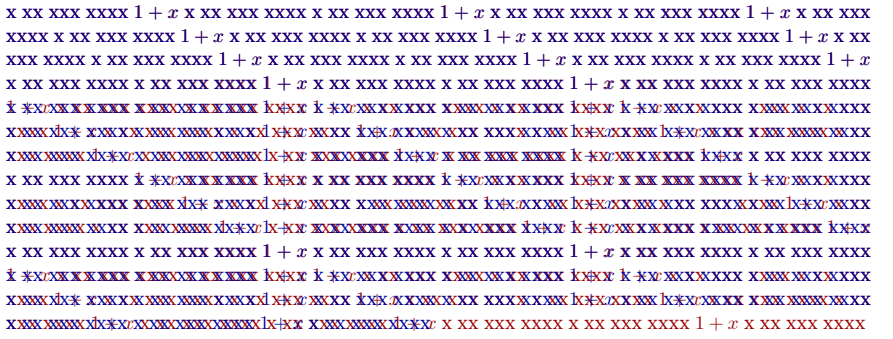
The second tweak is setting `\maththreshold` to some value. When set to for instance `40pt`, formulas that take less space than this will be wrapped in a `\hbox` and thereby will never break across a page.[3] Actually that second tweak has a variant so we have three tweaks! Say that we have this sample 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 below. 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.



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

3. A future version might inject severe penalties instead, time will learn.

$20^{20}+x_{20}^{20}$ 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

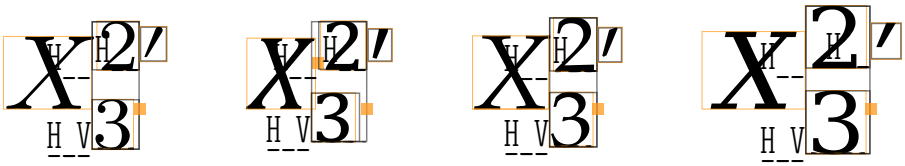
Flattening

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 uninterrupted 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 ε-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:

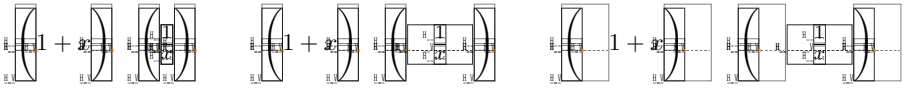
```
$ x = \left( \dorecurse{40}{\frac{x}{x+#1} +} x \right) $
```

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

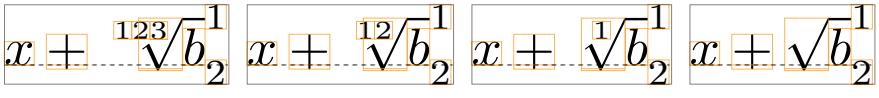
In traditional TEX a radical with degree is defined as macro. That macro does some measurements and typesets the result in four sizes for a choice. The macro typesets the degree in a box that contains the degree as formula. There is a less guesswork going on than with respect to how the radical symbol 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.[4] 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.

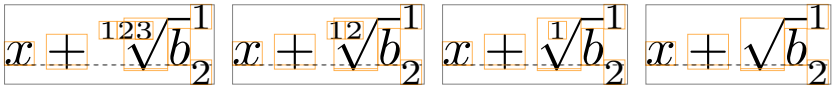
In OpenType the specification of the radical also includes spacing properties of the degree and that is why we have a primitive in LuaTEX that also handles the degree. It is what we used in ConTEXt MkIV. But . . . we actually end up with a situation that compares to the already discussed fraction: there 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 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
```

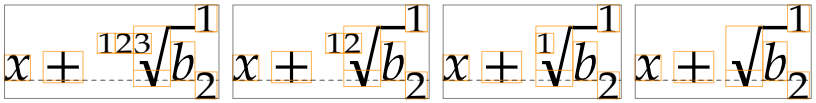


```
\setmathignore\Umathradicaldegreebefore 1
```

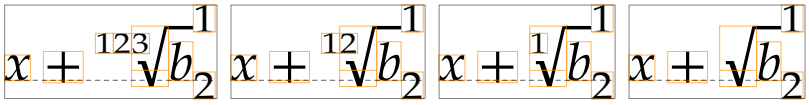
Latin Modern

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.

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



\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.

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

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:

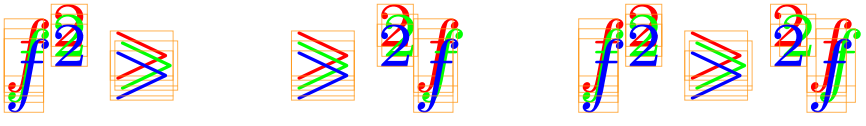
```
% \setmathignore \Umathspacebeforescript\zerocount % default
% \setmathignore \Umathspaceafterscript \zerocount % default

\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.[5]

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).

5. 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!

```
$
  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 17.

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
$
```

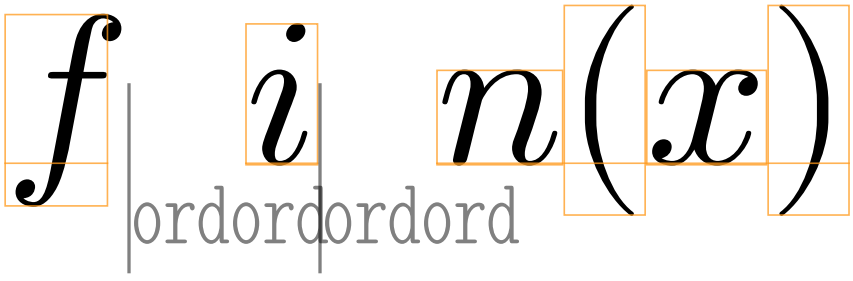
There was already a vcenter class available before we did this:

Text

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

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

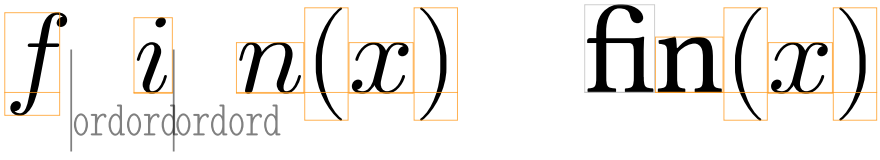
The result demonstrates that what looks like a word actually becomes three math atoms:



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:



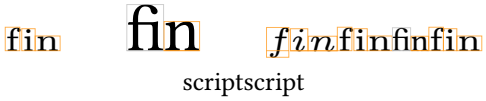
We even get a ligature which might be an indication that we're not using a math font which indeed is the case: the box is typeset in the regular text font.

```
\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:

text

script



scriptscript

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).[6]

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
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

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

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

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

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

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

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

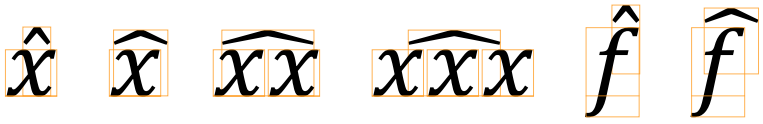
One of the other very sloped symbol is the integral, although most fonts have them more upright than tex has. Of course there are many variants of these integrals in a math font. Here we also have some font parameters that we can tune, which is what we do.

Accents

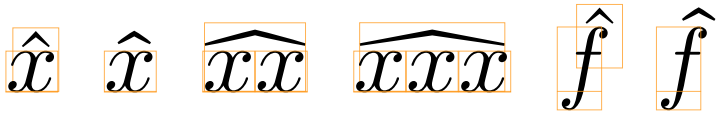
Accents are common in languages other than English and it's English that TEX was made for. Although the seven bit variant became eight bit handling accents never was sophisticated and one of the main reasons is of course that one could use pre-built composed characters. The OpenType format brought proper 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} $
```

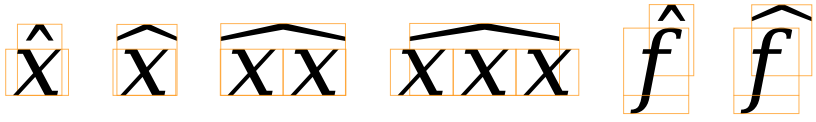
When looking at examples you need to be aware of the fact hat fonts can have been adapted in the goodie files.[7] So, for instance bounding boxes and such can differ from the original. Anyway, the previous code in Cambria looks as follows.



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 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.

7. 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.

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.[8]

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
   +\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.

8. In previous versions one could configure this per font but that has been dropped.

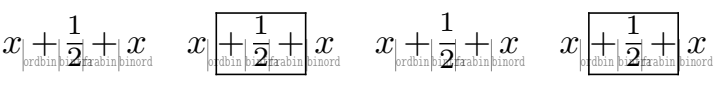
Dirty tricks

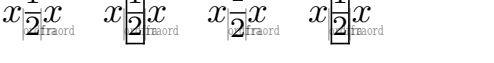
Once you start playing with edge cases you also start wondering if some otherwise complex things can be done easier. The next macro brings together a couple of features discussed in previous sections. It also uses two state variables: `\lastleftclass` and `\lastrightclass` that hold the most recent edge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

```
\tolerant\permanent\protected\def\NiceHack[#1]#:#2% special arg. 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=1500]{\MyTest{>}}            \blank
\scale[scale=1500]{\MyTest{+}}            \blank
\scale[scale=1500]{\MyTest{!}}            \blank
\scale[scale=1500]{\MyTest{+\frac{1}{2}+}}\blank
\scale[scale=1500]{\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.

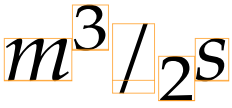




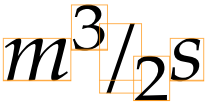
There are a few control options, like `\noatomruling` that can be used to prevent rules being applied to the next atom. We can use these in order to achieve more advanced alignment results, but discussing math alignments would demand many more pages than make sense here.

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.

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.

Once all the basics that have to do with proper dimensions, spacing, penalties and logic are dealt with, we will move on to the more high level constructs. So, expect more.

Hans Hagen & Mikael Sundqvist

Danlan type by Adriaan Goddijn

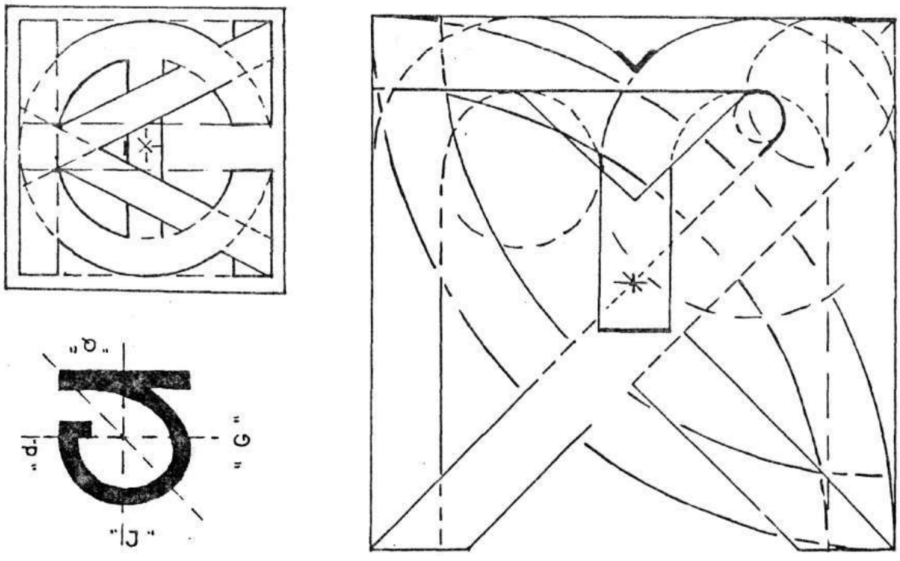
(and a salacious gnome)

The marketing mechanism of a Dutch Ebay version notified me that an article linked to a person with my surname had come up for sale. Usually that means an old book about the workings of electronic organs in the sixties by one Goddijn or history books about Catholic sociology by another Goddijn (both unrelated) but now another, more obscure Goddijn popped up as the maker of a slightly scandalous and weird colour pencil work depicting a garden gnome and his concubine.

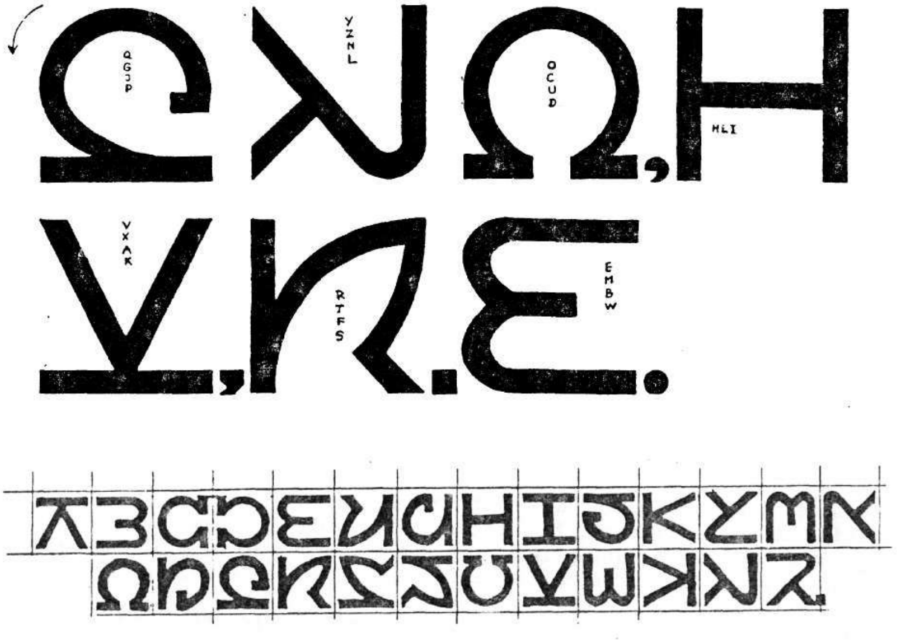
It's an art piece of cylindrical anamorphosis, meaning that the perspective of the image has been distorted to disguise an erotic image while enabling someone with a cylindrical mirror to recreate the 'hidden picture.'

To me, it seems a waste of valuable time and resources to create a rather silly gnome image which needs a special tool to see it. Nevertheless, Adriaan Goddijn (1925–2008), a painter and typesetter who studied in The Hague and taught at the Kopenhagen university, had fun doing it since he was fascinated by the chance reflection, in a reflective tin can, of a tobacco pouch.

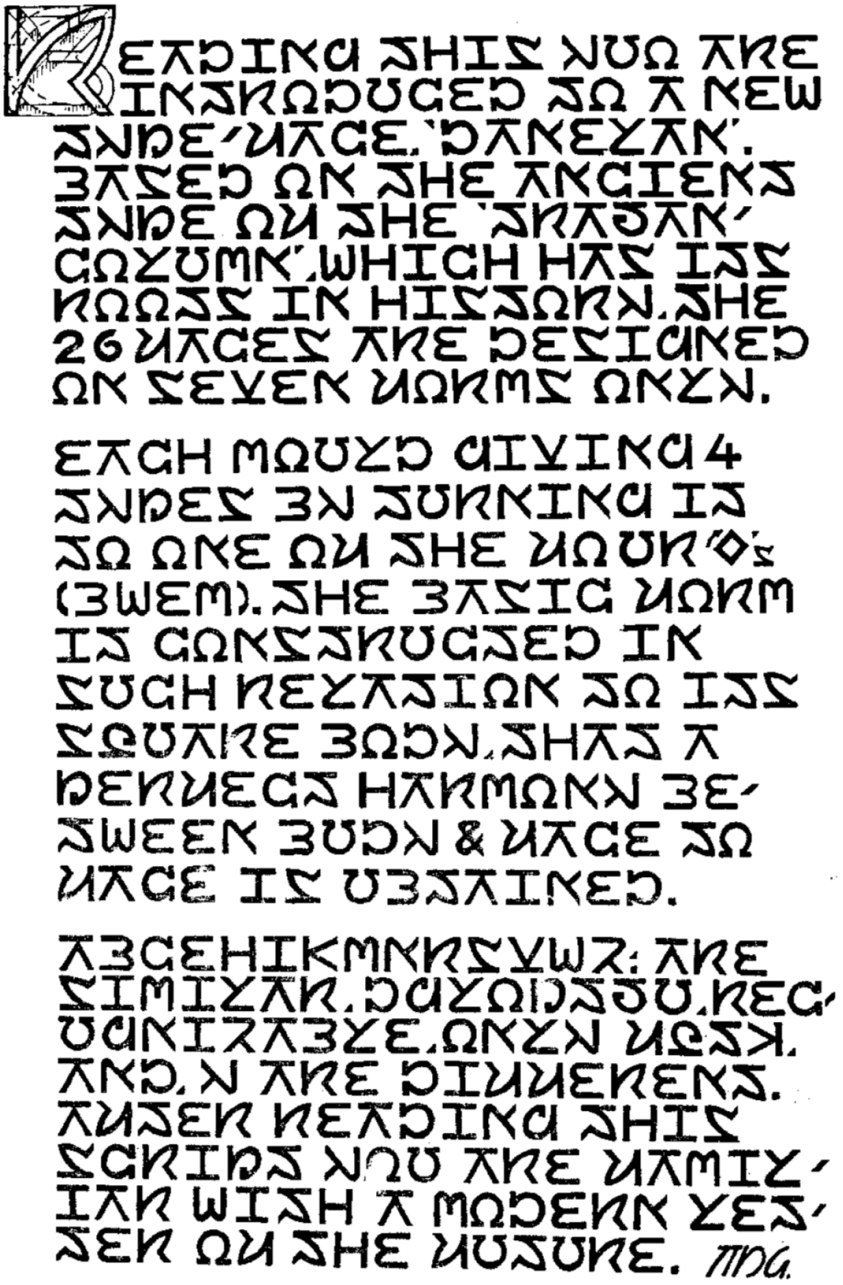
I went on and purchased the drawing, not for the image but to have a close look at the way the artist signed it, namely with his initials in a font he had created himself over the course of twenty years, in which every character of the alphabet is depicted using just seven glyphs. He named it DANLAN because he worked on it in the country of Denmark and the Dutch province of Friesland.



As a young man, he gold tooled the leather spines of books in his father's bookbinder's workshop and when he did his own book binding later in life he missed having the vast collection of handle letters he had had at his disposal at his father's place. He wanted to engrave his own letters but decided to limit the number of glyphs to save time. He never got around to actually engrave but he did design a font which needs an absolute minimum of glyphs, inspired by the Roman Capitalis Quadrata (upper caps) but simplified. Six of his glyphs can depict four characters each and the seventh can depict two, depending on how they are turned, left, right, upside down. Each glyph has the same width and height, fitting in a square so any side of the square can be the bottom line.



It takes a little bit of practice but the characters are just familiar enough to our eyes to be able to quickly recognize the words:



[bron o.a. Arthesis, mededelingenblad van de Stichting Art et Mathesis, Jaargang 5, nummer 2, april 1991]

Frans Goddijn

Danlan type by Adriaan Goddijn

(quick font hack)

Abstract
When Frans Goddijn first showed me the Danlan font article in September 2019, I immediately thought that it would be fun to play with those letters a bit in TEX and MetaPost.

But then the almost inevitable thing happened that so often happens to me: I got distracted by other things, and forgot about Danlan completely. Until this spring, when Frans reminded me that I had promised an article for the Maps. This is that promised article: it will show what a few days playing around with a specification and MetaPost, FontForge, and ConTEXt got me. I have not created a complete font by any means, but it is just enough of one to show off a little bit and document how the creation process worked out for me.

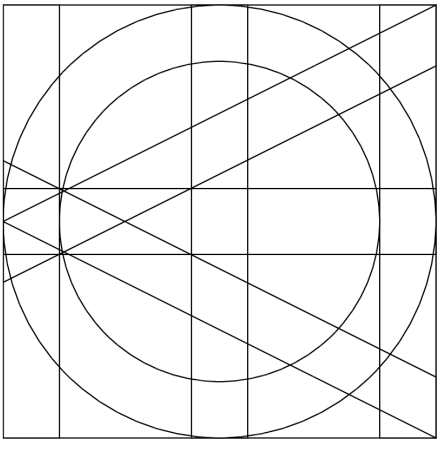
Preliminaries

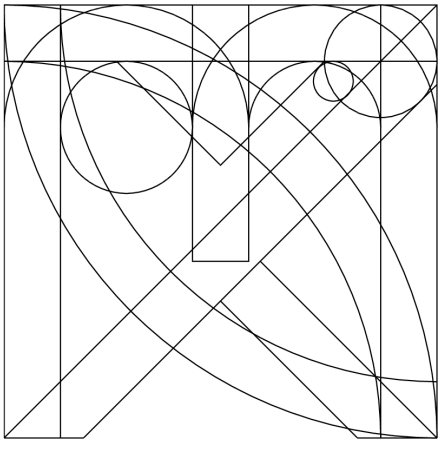
My first attempt was to bitmap-trace the seven glyphs from the demonstration image straight into FontForge. I tried that first because at the time I did not quite understand the design helper images. The plan was to have the glyphs drawn in FontForge, then convert back to MetaPost input format afterwards for playing around with them.

However, the results were very underwhelming. The resolution of the bitmaps was just not high enough for a clean trace. So, just using FontForge did not work out. But as I originally wanted to play with MetaPost anyway that was not a big deal. In this early stage I was clearly trying to cut too many corners.

In preparation for a second attempt, I decided to recreate the two design images in MetaPost, and then continue work from those. I should probably apologize for the sloppiness of the MetaPost code shown below, but I won't! This messy stuff is how I typically work, and then if something needs publishing, I clean it up after its functionality is already at 100%.

Here are the two drawings I created:





Not quite the same as the example drawings, but that is OK, as these contain all the lines that I needed. The few extra lines and curves are fine, I just needed to recreate the image to understand the design. They were never intended to be used as anything else than a quick reference for myself, and the MetaPost code clearly shows that.

This is the left image; this one is used for H and I as well as A, K, V, and X:

```
d = 130;
w = cosd(22.5);
e = d * 1/(w*w);
path leftup;
pickup pencircle scaled 3;
draw unitsquare scaled 1000;
leftup = (0, 500-(d/w))--(1000,1000-(d/w))--(1000,1000)--(0,500)--cycle;
draw leftup;
draw leftup reflectedabout ((0,500),(1000,500));
draw (0,0)--(d,0)--(d,1000)--(0,1000)--cycle;
draw (500-(d)/2,0)--(500+(d)/2,0)--(500+(d)/2,1000)--(500-(d)/2,1000)
                                                        --cycle;
draw (1000-d,0)--(1000,0)--(1000,1000)--(1000-d,1000)--cycle;
draw (0,500-e/2)--(1000, 500-e/2)--(1000, 500+e/2)--(0,500+e/2)--cycle;
draw fullcircle scaled 1000 shifted (500,500);
draw fullcircle scaled (1000-2*d) shifted (500,500);
```

The interesting things in this drawing are:

- `d`, which is the width of the vertical bars. This becomes an actual font meta-ness parameter (the only one).
- `w`, which is a MetaPost shortcut. It makes the diagonals the same apparent width as the vertical bars.
- `e`, which is the width of the horizontal middle bar. The definition is a bit odd, but this most closely matches Adriaan Goddijn's drawing.

And the units are simply set up for ease of use.

Here is the right image, which is used for all the other letters in the alphabet:

```
v = cosd(45);
save diam,x,y,p;
path p[];
p1 = ((0+d/v,0)--(1000,1000-(d/v)));
z5 = ((1000,0)--(0,1000)) intersectionpoint p1;
z6 = ((1000-d/v,0)--(0,1000-d/v)) intersectionpoint p1;
z1 = (d+d, 1000-d); z1' = (x1 + 1000,y1 - 1000);
z2 = (1000-d-d,y1); z2' = (x2 - 1000,y2 - 1000);
z3 = (z1--z1') intersectionpoint (z2--z2');
z4 = (z1--z1') intersectionpoint ((0,0)--(1000,1000));
diam = arclength (z3--z4);
draw (0,0)--(0,1000)--(1000,1000)--(1000,0);
draw (0,0)--(d/v,0);
draw (1000,0)--(1000-d/v,0);
draw quartercircle scaled 2000;
draw quartercircle scaled (2000-2*d);
draw (quartercircle scaled 2000) rotatedaround((500,500), 180);
draw (quartercircle scaled (2000-2*d)) rotatedaround((500,500), 180);
draw (0,0)--(1000,1000);
draw (0+d/v,0)--(1000,1000-(d/v));
draw (d,0)--(d,1000);
draw (1000-d,0)--(1000-d,1000);
```

```
draw (0,1000-d)--(1000,1000-d);
draw (1000,0)--z5;
draw (1000-d/v,0)--z6;
x7 = 500-d/2;
x7'= 500+d/2;
y7'= 1000 -(x7'/2);
draw (x7, 1000)--(x7, y5)--(x7',y5)--(x7',1000);
draw halfcircle scaled x7' shifted (x7'/2,y7');
draw (halfcircle scaled x7' shifted (x7'/2,y7')) shifted (x7,0);
draw fullcircle scaled (x7'-2*d) shifted  (x7'/2,y7');
draw (halfcircle scaled (x7'-2*d) shifted (x7'/2,y7')) shifted (x7,0);
draw z1--z3--z2;
draw fullcircle scaled (2*d) shifted  (1000-d, 1000-d);
draw fullcircle scaled diam shifted (x2+v*(diam/3.1415), 1000-d-diam/2);
```

That one is a bit more involved. By far the most interesting object in that drawing is that little circle in the top right. Deducing its diameter and (especially) its location was a bit tricky. I am still not sure I have done it completely right, but the current definition produces a result that – I think – is good enough.

Implementation

With the two design images done, it became a simple exercise to code the seven main glyphs in MetaPost. For example, here is the basic definition of one of them:

```
def letter_k =
  begingroup
  save e,leftup,leftdown,vb,vt;
  path leftup, leftdown;
  e = d * 1/(w*w);
  leftup = (0, 500-(d/w))--(1000,1000-(d/w));
  leftdown = leftup reflectedabout ((0,500),(1000,500));
  vb = 500-e/2;
  vt = 500+e/2;
  z1 = ((d,0)--(d,1000)) intersectionpoint ((0,vb)--(1000,vb));
  z2 = ((0,vb)--(1000,vb)) intersectionpoint ((0,500)--(1000,0));
  z3 = ((d,0)--(d,1000)) intersectionpoint ((0,vt)--(1000,vt));
  z4 = ((0,vt)--(1000,vt)) intersectionpoint ((0,500)--(1000,1000));
  z5 = leftup intersectionpoint leftdown;
  (0,0)--(d,0)--z1--z2--(1000,0)--(1000,(d/w))--z5--(1000,1000-(d/w))--
     (1000,1000)--z4--z3--(d,1000)--(0,1000)--cycle
  endgroup
enddef;
```

In total, there were seven of these definitions. Each of these is called two times: once for a filled-in style, and once for an outline style:

```
beginchar(107, "k")
fill letter_k;
endchar;

beginchar(75, "K")
fill letter_k withcolor white withpen pencircle scaled ministrokew;
draw letter_k withpen pencircle scaled strokew;
currentpicture := currentpicture shifted(strokew/2, strokew/2);
endchar;
```

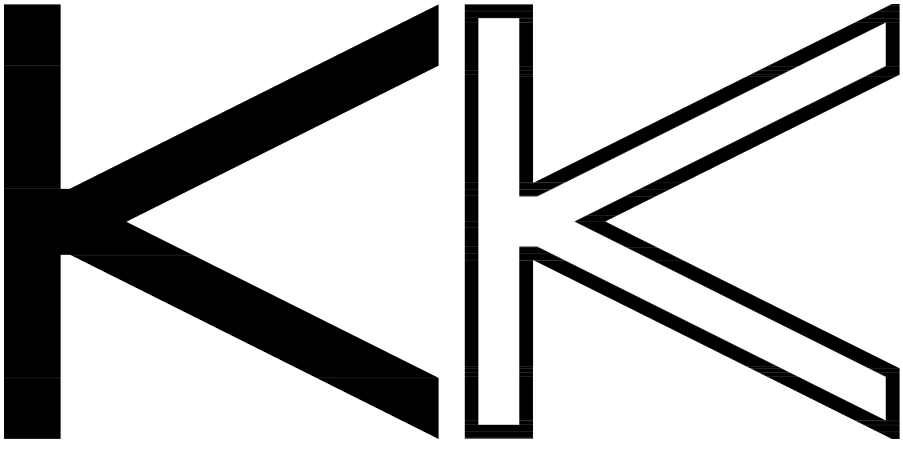
Finally, there is a bit of a preamble to that file needed:

```
d := stem;
w := cosd(22.5);
v := cosd(45);
ministrokew := (stem/11); % just to give width to the eraser

linejoin := mitered;
miterlimit := infinity;
linecap := butt;

def beginchar(expr c, s) =
  if string s:
    outputtemplate := "%j-%c-"&s&".%o";
  else:
    outputtemplate := "%j-%c.%o";
  fi
  beginfig(c)
enddef;

def endchar =
  currentpicture := currentpicture scaled 0.8;
  currentpicture := currentpicture shifted (lsidebearing,0) ;
  endfig
enddef;
```

All those macros were saved in a MetaPost file, and then a driver file is used to set the small amount of variables:

```
stem = 130;
lsidebearing = 70;
strokew = (stem/4);

input danlan-design;

end.
```

Processing one of these driver files produces the raw EPS images of the base glyphs. For example, here are the ‘k’ and ‘K’ versions in the regular width:



At that point, the ‘real’ work was done. What was left was to import to glyphs in FontForge, do some visual manipulation like adding the rotated versions, adding points at extrema, rounding all points to integers, and setting the right sidebearings.

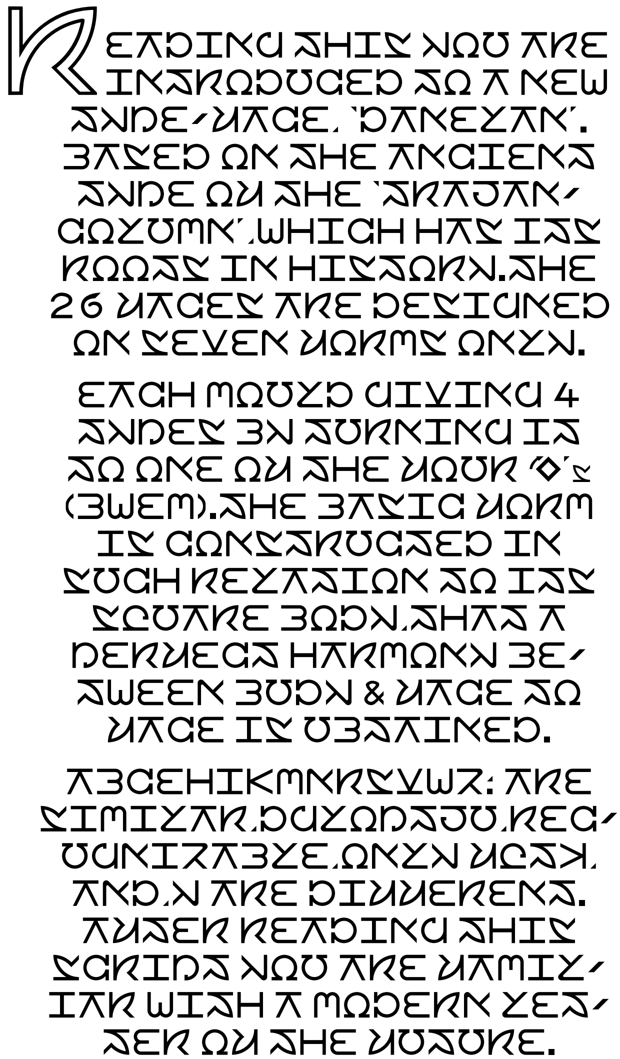


Figure 1. Regular version.

Figure 2. Bold version.

Final touches and result

There are a few glyphs in the demonstration text that were not based on the design, like the punctuation marks, the digits, and the rotation symbol. Those were simply created in FontForge directly for the 'regular' version of the font. My capital letters are less elaborate than single 'R' initial from the demonstration text, but I wanted to create all of the uppercase letters and it was not obvious from the demonstration how many and which of the helper lines should be added as the decorative backdrop to the initial, so I chose to skip all of them.

```
\definefont[danlannormal][Danlan at 12pt]
\definefont[danlaninit]  [Danlan at 40pt]
\definefont[danlanx]     [Danlan at 8pt]
```

After generating an OpenType font from FontForge, it was time to prepare a TEX version of that textual demonstration:

```
\danlannormal

\setbox1=\hbox
  {\kern -18pt
   \smash{\hbox{\danlaninit R}}}

\blank[2*line]
\setuplocalinterlinespace
  [line=14pt]

\startlines
\dontleavehmode %
\kern\wd1 eading this you are
\dontleavehmode %
\box1 introduced to a new
type-face, `danelan'.
based on the ancient
type of the `trajan-
column',which has its
roots in history.the
26 faces are designed
on seven forms only.
each mould giving 4
types by turning it
to one of the four @'{\danlanx s}
(bwem).the basic form
is constructed in
such relation to its
square body,that a
perfect harmony be-
tween budy & face to
face is ubtained.

abcehikmnrsvwz: are
similar,dgloptju,rec-
ugnizable,only fqtx,
and,y are different.
after reading this
script you are famil-
iar with a modern let-
ter of the future.
\stoplines
```

The text above in a faithful representation of the demonstration image. Interestingly, it has a number of typos. The typeface name is actually OK (it was initially named 'danelan' instead of 'danlan') but there are a few mistakes with u instead of o, and in the alphabetic list at the end, the 't' is mentioned twice!

The typeset result is in figure 1.

Using a slightly different driver file with a wider stem, it was possible to create a bold version of the font as well. This one only has the 52 main glyphs, and its effect can be seen in figure 2.

Afterthoughts

Having spent two days on the fonts in the end creating something that is of extremely limited use, I have to wonder whether this was worthwhile. The chance of anyone extending my MetaPost code to a complete, usable font family seems infinitesimally small.

On the other hand, it did result in this article and perhaps most importantly: I had a lot of fun playing around!

Taco Hoekwater

Finding all intersections of paths in MetaPost

Abstract
In this article we will discuss different ways to implement macros to find all intersections of paths in MetaPost. We will first work out some rather simple ideas, providing partial solutions on the macro level. We then show by an example how the `intersectiontimes` macro works, and describe how it was extended in the engine by Hans Hagen.

Introduction

In MetaPost the fundamental way to find intersections of paths is to first find the times of the intersections by running

```
p intersectiontimes q
```

If the paths `p` and `q` intersect, this will return a pair of times `(t1,t2)` such that the path `p` at time `t1` intersect the path `q` at time `t2`. Often, *but not always*, this will return the first intersection point of the paths `p` and `q`, in the sense that `t1` will be the smallest time along `p` for which the paths intersect. The uncertainty comes from the implementation, that relies on a bisection algorithm for finding intersection points of Bézier curves. The algorithm is inherited from MetaFont, and described in detail in [Knu86], but we will also illustrate it later.

In Figure 1 we have drawn a path `p` in blue. Along this path we have placed ten different circles, each playing the rôle of `q`. We run `p intersectiontimes q` and draw with a green dot the point that correspond to the time returned. For the first four circles and for the tenth circle, `intersectiontimes` returns the second intersection point. For the fifth to ninth circle it returns the first intersection point.[1]

```
\startMPcode[instance=doublefun]
numeric n ; n:=10 ;
path p,q ;

p = (0,0) .. (6cm,4cm){dir 20} .. {up}(6cm,0) ;
drawarrow p withcolor darkblue ;

for i = 1 upto n :
  q := fullcircle scaled 30bp shifted point (i/(n+1)) along p ;
  drawarrow q withcolor darkred ;
  tone := xpart (p intersectiontimes q) ;
  drawdot point tone of p withpen pencircle scaled 6bp
    withcolor darkgreen ;
endfor ;
\stopMPcode
```

1. We will often work with pairs of paths `p` and `q`, and since the order will be important, we will always draw the first one (`p`) in blue and the second one (`q`) in red. Moreover, green dots will be used for the built-in macros (or ones written by others). We use orange dots to show our constructed points.

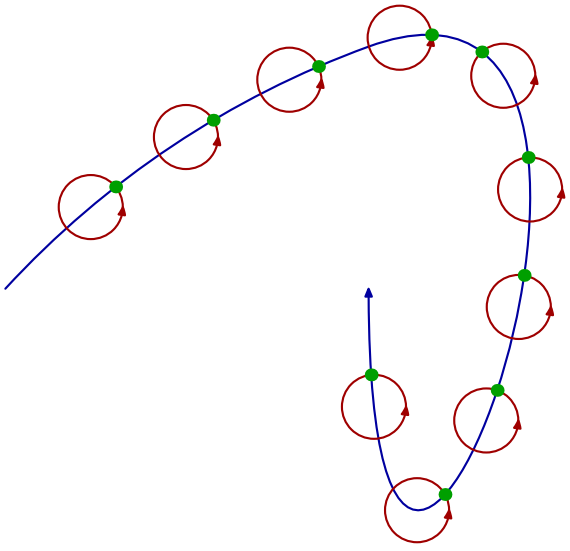


Figure 1.

This is sometimes inconvenient. For example, if we want to cut one path after it intersects another path for the first time (usually done with `cutafter`), or if we want to find all intersection points of two paths (how to loop?), and it would be desirable to have a more reliable algorithm for finding the first intersection point (or all intersection points) of two paths. We are not alone in thinking so[2].

The macro `cutbefore` is used as `p cutbefore q`; it is supposed to return the part of the path `p` that remains after the *first* intersection of `p` and `q`. It often does, but in the MetaPost manual [Hob20] we can read that `p cutbefore q` is equivalent to

```
subpath (xpart(p intersectiontimes q), length p) of q
```

As `intersectiontimes` is used, we will not always get the first intersection, and therefore the user must be careful to see that the result is the intended.

```
\startMPcode[instance=doublefun]
path p, q;
p := (0,0) -- (5cm,0) ;
q := fullcircle rotated -20 scaled 2cm xshifted 2.5cm ;
pair b ; b := (p intersectiontimes q) ;
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;
drawdot point xpart b of p withpen pencircle scaled 6bp
  withcolor darkgreen ;
\stopMPcode
```

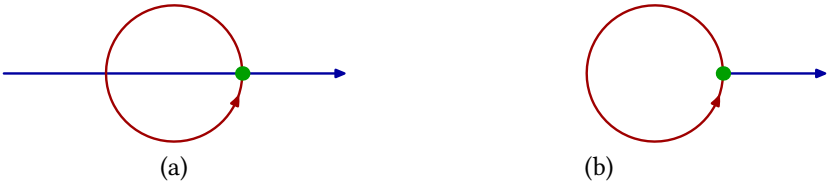


Figure 2. (a) For this pair of curves `intersectiontimes` returns the times for the second intersection point. (b) As a consequence `cutbefore` cuts at the time that `intersectiontimes` returned.

2. See for example the questions https://tex.stackexchange.com/q/79256/52406 and https://tex.stackexchange.com/q/180510/52406.

MetaPost paths are piecewise built with help of Bézier curves, polynomials of degree three. For us it is convenient to think of a path p as a parametrized path where the parameter t, which we usually call time, runs over a certain interval with endpoints 0 and length p. For the path p in figure 1 it holds that length p equals 2; the three points used in the definition correspond to time 0, time 1 and time 2. We call the parts between the points the *segments* of the path. Each segment corresponds to a unit time interval.

In the search for solutions

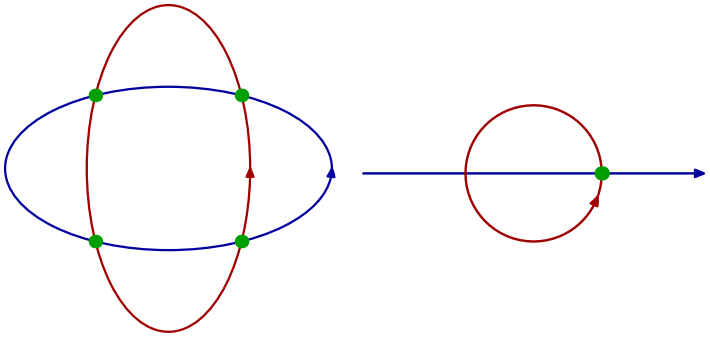
At TEX StackExchange, in an answer[3] to the question mentioned before, we find a suggestion on how to find all intersection points of two paths. It is based on a suggestion on the MetaPost mailing list [Hen08].

```
\startMPcode[instance=doublefun]
path p, q, r ;
p := fullcircle xscaled 144 yscaled 72 ;
q := fullcircle xscaled 72 yscaled 144 ;
r := p ;
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;
n := 0 ;
forever :
  r := r cutbefore q ;
  exitif length cuttings = 0 ;
  r := subpath(epsilon, length r) of r ;
  z[n] = point 0 of r ;
  drawdot z[n] withpen pencircle scaled 6bp
    withcolor darkgreen ;
  n := n + 1 ;
endfor;
\stopMPcode
```

As you can see in Figure 3(a) the intersection points are found. In Figure 3(b) we use the same method, but with the paths p and q changed into

```
p:= (0, 0) -- (5cm, 0) ;
q:= fullcircle rotated -20 scaled 2cm xshifted 2.5cm ;
```

only the right intersection point is found, which is not a surprise to you by now.



(a) (b)

Figure 3.

3. https://tex.stackexchange.com/a/79332/52406

The document [Thu17] contains a lot of beautiful MetaPost graphics to get inspiration from. It contains also a nice discussion on the difficulty of finding all intersection points of two curves (Sections 9.4.1 and 9.4.1), concluding with essentially the same suggestion as the one given above.

To find the first intersection time, a first try

One way to find all intersection points of two paths could be to first have a method to find the first intersection point. We will try to find the first intersection point by repeating the use of `intersectiontimes` on smaller and smaller subpaths.

We illustrate the idea by looking at the line and the circle again. First we use `p intersectiontimes q` to find one (any) intersection point, see Figure 4(a). Then we cut away the part of the path `p` that comes after that intersection. This is shown in Figure 4(b). In fact, we cut just a bit more, to be sure not to arrive at the point we just got. Once that is done, we repeat, see Figure 4(c), where a second intersection is found. In our example, this is the last one, since once we cut again, the paths do not intersect any more, see Figure 4(d). The algorithm returns the time of the last intersection point that was found, in our case the time of the orange dot marked in Figure 4(c).

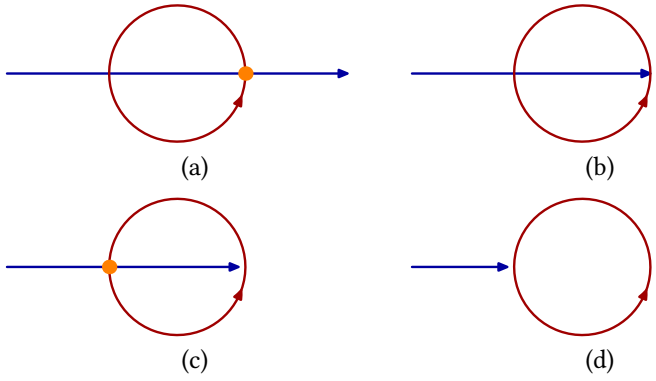


Figure 4.

We implement this algorithm below.

```
\startMPdefinitions{doublefun}
tertiarydef p firstintersectiontimetryone q =
  begingroup;
  save tres, tmpt, tmpp, tmpl ;
  pair tres ; tres := (-1, -1) ;
  pair tmpt ; tmpt := (length p, 0) ;
  path tmpp ; tmpp:=p;
  numeric tmpl ;
  forever :
    tmpt := tmpp intersectiontimes q ;
    exitif xpart tmpt < 0 ;
    tmpl := arclength subpath(0, xpart tmpt) of tmpp ;
    tres := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(0, (xpart tres) - epsilon) of p ;
  endfor ;
  tres
  endgroup
enddef ;
\stopMPdefinitions
```

It might be helpful to make a few comments. The `tmpl` variable might at a first glance seem superfluous. But the `tmpt` variable calculates the time along `tmpp`. It doesn't necessarily hold that this time is the same as the time along the original path `p`. So, we work with lengths instead, since they do not change with parametrizations. We first use `arclength` set `tmpl` to the length of the part of the temporary path `tmpp` before the cut. Then we use `arctime` to calculate the time on the original path `p` that corresponds to this length. This is probably a stupid thing to do numerically, since there are several points where inaccuracies might be introduced, but at least it seems to work for simple examples. When the conversion is done we update the temporary path `tmpp`, and cut away an `epsilon` (that is $1/65536$) of it in the end, not to get stuck in an infinite loop.

We test this definition on the different combinations of `p` and `q` from Figure 1. This is done with the following code[4], and the result is shown in Figure 5.

```
\startMPcode[instance=doublefun]
numeric n, standardfirst, ourfirst ; n := 10 ;
path p,q ; p := (0, 0) .. (6cm, 4cm){dir 20} .. {up}(6cm, 0) ;
drawarrow p withcolor darkblue ;

for i = 1 upto n :
  q := fullcircle scaled 30bp shifted point (i / (n + 1)) along p ;
  drawarrow q withcolor darkred ;
  standardfirst := xpart (p intersectiontimes q) ;
  drawdot point standardfirst of p withpen pencircle scaled 6bp
    withcolor darkgreen ;
  ourfirst := xpart (p firstintersectiontimetryone q) ;
  drawdot point ourfirst of p withpen pencircle scaled 4bp
    withcolor "orange" ;
endfor;
\stopMPcode
```

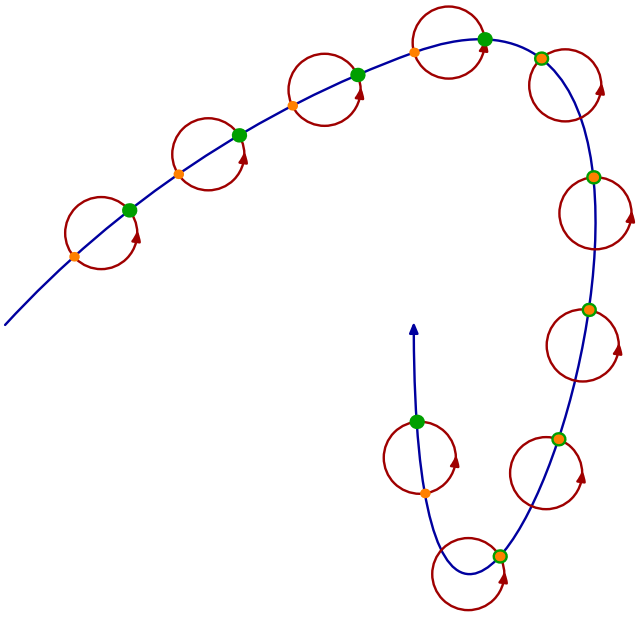


Figure 5.

4. Note that we write `darkred` and `darkblue` but `"orange"`. The orange color is undefined in MetaPost/MetaFun, so we borrow the definition from ConTEXt.

As you can see, the new macro returns the first intersection points in all ten cases. This could have been the happy end, but, as we will see, it is not. Not to be too crestfallen at this point, we show in Figure 6 some examples where the algorithm works.

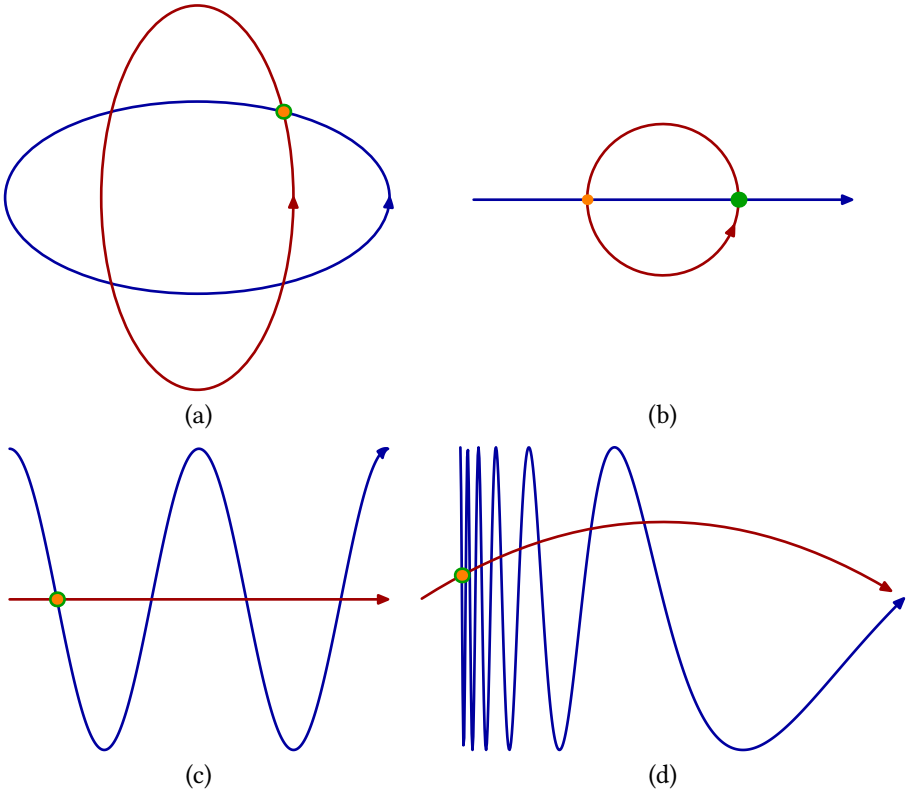


Figure 6.

To find the first intersection time, a second try

If the paths p and q start at the same point we have a problem. Our algorithm gets stuck in an infinite loop. This happens for example for the paths

```
p := (0, 0){dir 45} .. (5cm, 1cm) ;
q := (0, 0) -- (5cm, 0cm) ;
```

The standard solution with `p intersectiontimes q` returns the intersection at time zero, and we have marked that point in Figure 7.

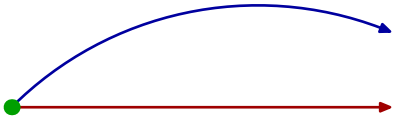


Figure 7.

To overcome this problem we introduce a test to check if we have ended up at the beginning of the path. We introduce a boolean `tzero` that we set to `true` if the intersection time found is less than a certain value, which we have chosen to be `epsilon`.

```
\startMPdefinitions{doublefun}
tertiarydef p firstintersectiontimetrytwo q =
```

```
  begingroup ;
  save tres, tmpt, tmpp, tmpl, tzero ;
  pair tres ; tres := (-1, -1) ;
  pair tmpt ; tmpt := (length p, 0) ;
  path tmpp ; tmpp := p ;
  numeric tmpl ;
  boolean tzero ; tzero := false ;
  forever :
    tmpt := tmpp intersectiontimes q ;
    exitif xpart tmpt < 0 ;
    if xpart tmpt < epsilon :
      tzero := true ;
      exitif true ;
    fi
    tmpl := arclength subpath(0, xpart tmpt) of tmpp ;
    tres := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(0, (xpart tres) - epsilon) of p ;
  endfor ;
  if tzero :
    tmpt
  else :
    tres
  fi
  endgroup
enddef ;
\stopMPdefinitions
```

In Figure 8(a) we see that our new approach works well, the first point is found. In Figure 8(b) we have just included an example where the paths intersect at the endpoint of p. No problem!

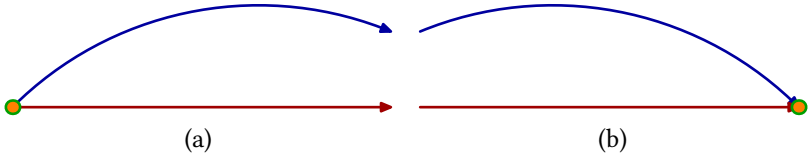


Figure 8.

We will not refine the algorithm for finding the first intersection time further here, but leave only some comments.

The constant `epsilon` is the *time step* we take from a previously found intersection time. If the path p consists of many points (this is the case for the case of the $\sin(1/x)$ path above, where very small sampling step is chosen) then this means that two points in the path can lie very close to each other. The time step between them is still one, so an `epsilon` time step away along the path might result in indistinguishable points, and that might result in an infinite loop. One way to solve such an issue could be to add a new variable to the macro that replaces `epsilon`. That would require more from the user; for example it would be necessary to have the sampling step in function graphs in mind while working with intersections.

Instead of removing an `epsilon` time part of the path, one could convert to lengths and remove an `epsilon` length of the path. That would lead to a potential lost of accuracy in the calculations, but it could at the same time be easier for the user to set a threshold in terms of length instead of time.

To find all intersection times, a first try

With the time for the first intersection at hand, it is in principle straightforward to find all intersection times. We illustrate with the line and the circle again. In Figure 9(a) we have marked the first intersection point. This time we cut away the part of the path p *before* the intersection point. This is done in Figure 9(b). Again, we need to cut a bit more, not to end up with an infinite loop. So, in Figure 9(c) we have cut just a bit more, and the next intersection is found. Finally, in Figure 9(d) we have cut away so much that the paths do not intersect anymore. We have found all intersections.

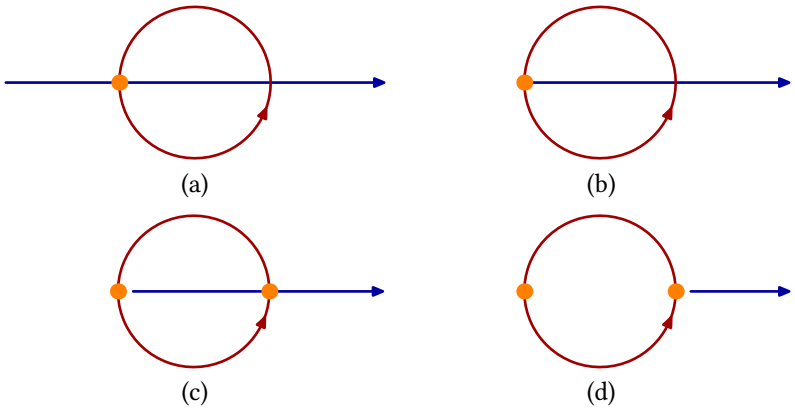


Figure 9.

We try with the following algorithm.

```
\startMPdefinitions{doublefun}
tertiarydef p allintersectiontimestryone q =
  begingroup ;
  save tmpt, tmpp, tmpptmp, tmpl, it, n ;
  pair tmpt ;
  path tmpp, tmpptmp ; tmpp := p ;
  numeric tmpl ;
  pair it[] ;
  numeric n ; n := 0 ;
  forever :
    tmpt := tmpp firstintersectiontimetrytwo q ;
    exitif xpart tmpt < 0 ;
    tmpptmp := subpath(xpart tmpt, length tmpp) of tmpp ;
    tmpl := arclength p - arclength tmpptmp ;
    it[incr n] := (arctime tmpl of p, ypart tmpt) ;
    tmpp := subpath(xpart tmpt + eps, length tmpp) of tmpp ;
  endfor ;
  topath(it, --) % it[1] -- it[2] -- ... -- it[n]
  endgroup
enddef ;
\stopMPdefinitions
```

We first test it on the examples from Figure 6. The code for the ellipses is shown below, the other are similar. As you can see in Figure 10 it works in all cases.

```
\startMPcode[instance=doublefun]
path p ; p := fullcircle xscaled 144 yscaled 72 ;
path q ; q := fullcircle xscaled 72 yscaled 144 ;
```

```
drawarrow p withcolor darkblue ;
drawarrow q withcolor darkred ;

path b ; b := p allintersectiontimestryone q ;

for i = 0 upto length b :
  drawdot point xpart (point i of b) of p
    withpen pencircle scaled 6bp
    withcolor "orange" ;
endfor ;
\stopMPcode
```

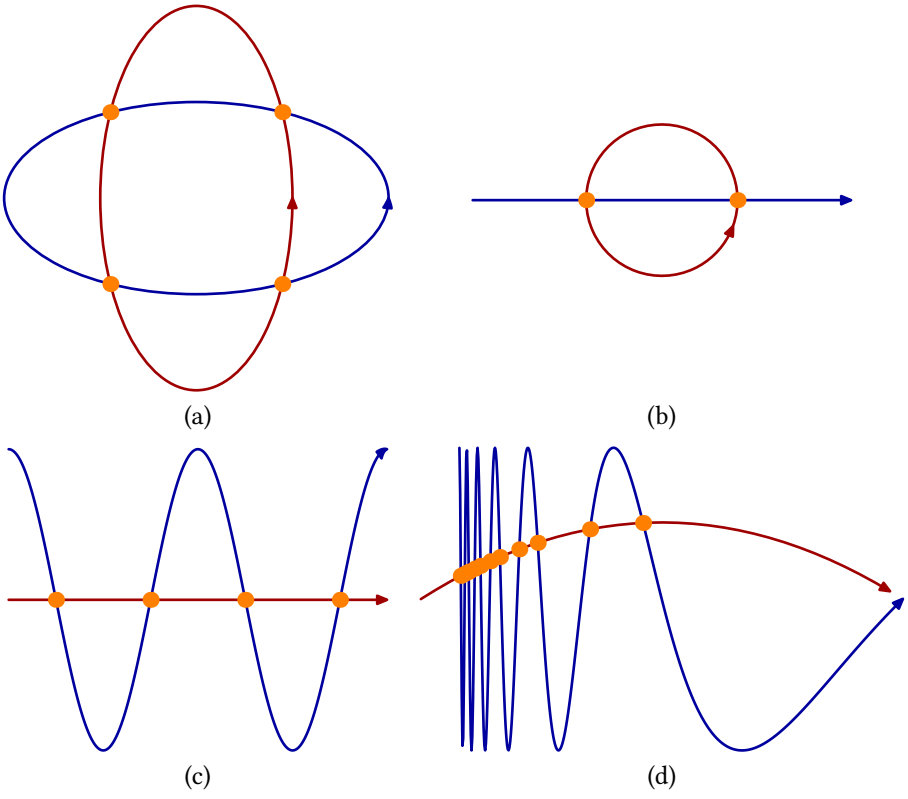


Figure 10.

We also try it on the paths from Figure 5, the result can be seen in Figure 11. Again we get all intersections.

The name of this section suggests that this implementation has problems, and indeed it has. This time it is not the first point of the path p that is the problem, but the last.

To find all intersection times, a second try

The macro `allintersectiontimestryone` works fine for the paths Figure 8(a) that intersect at the first point of p, but it hangs with an infinite loop for the paths in Figure 8(b). The reason is that the paths intersect at the endpoint of p. We must, similarily as we did when we found the first intersection point, check if we are at the end of the path. We rewrite the macro like this.

```
\startMPdefinitions{doublefun}
tertiarydef p allintersectiontimes q =
  begingroup ;
  save tmpt, tmpp, tmpptmp, tmpl, tmpL, it, n, a, cuttime ;
  pair tmpt, it[] ;
  path tmpp, tmpptmp ;
  numeric tmpl, tmpL, n ; n := 0 ;
  tmpp := p ;
  tmpt := tmpp firstintersectiontimetrytwo q ;
  forever:
    exitif xpart tmpt < 0 ;
    exitif length(tmpp) = 0 ;
    tmpptmp := subpath(xpart tmpt, length tmpp) of tmpp ;
    tmpl := arclength p - arclength tmpptmp ;
    it[incr n] := (arctime tmpl of p, ypart tmpt) ;
    tmpL := arclength(subpath(0, xpart tmpt) of tmpp) + eps ;
    if tmpL > arclength tmpp :
      exitif true ;
    else :
      cuttime := arctime (tmpL) of tmpp ;
      tmpp := subpath(cuttime, length tmpp) of tmpp ;
    fi ;
    tmpt := tmpp firstintersectiontimetrytwo q ;
  endfor ;
  topath(it, --)
  endgroup
enddef ;
\stopMPdefinitions
```

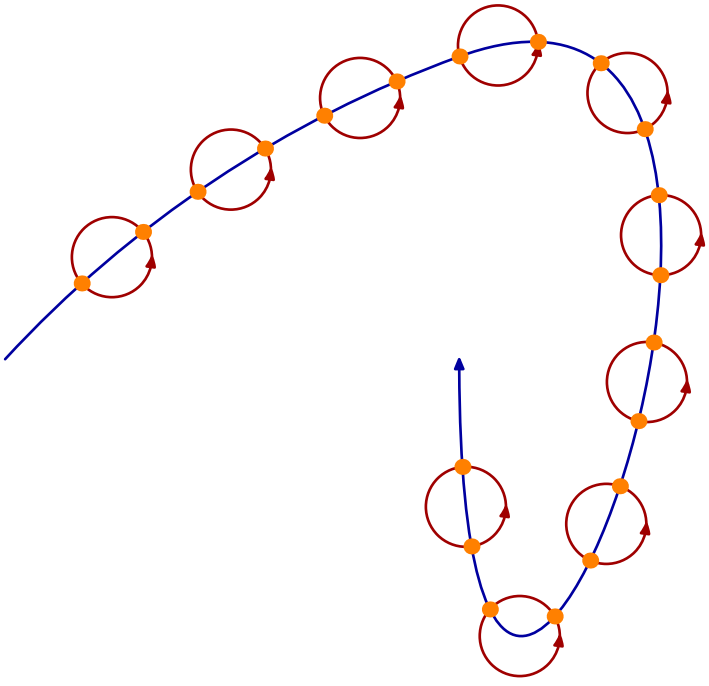


Figure 11.

Here we have introduced `tmpL` to check if we are at the endpoint of the path `p`. As we see in Figure 12 we now catch intersections both at the beginning and at the end of the path `p`.

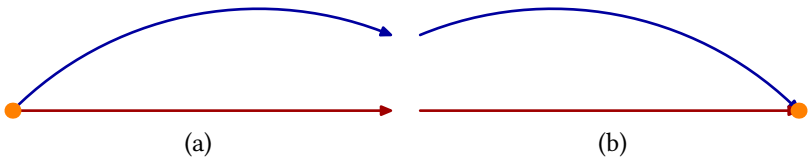


Figure 12.

We could continue here with refinements. We could for example condition on if `p` is a loop or not, we could tune the size of the steps[5] with which we jump ahead after finding an intersection point, and so on. But our final way of solving the problem will be different, so we stop here. You have a macro that works well in most cases.

Returning to the `intersectiontimes` algorithm

I was discussing the problem of finding all intersection points with Hans Hagen, and he had a slightly different idea than the one we have described above. Instead of first iterating to get the first intersection point and then to cut and iterate, Hans wanted to tweak the existing `intersectiontimes` algorithm in order to find all intersection times. Let us first discuss the `intersectiontimes` algorithm in more detail.

We recall that a path in metapost can be thought of as an array[6] of points and control points that together describe the segments that build the path. In Figure 13(a) we have drawn a path with three segments and labeled the four points that define it. Each segment correspond to a unit time interval, and it can mathematically be described by a Bézier curve. In Figure 13(b) we have emphasized the segment between time 2 and time 3, that is `subpath (2,3) of p`. We have also drawn the two control points and control lines of this segment.

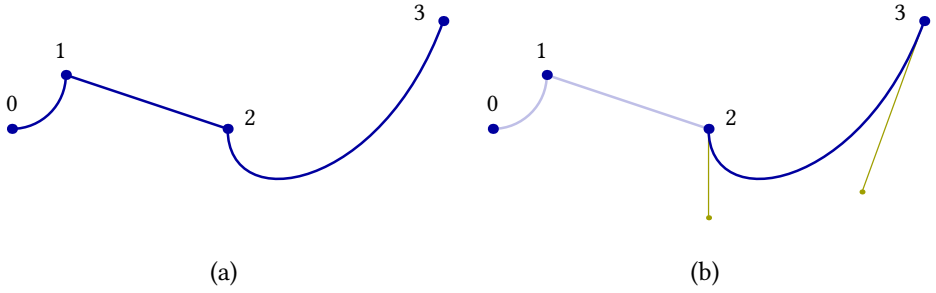


Figure 13.

The `intersectiontimes` macro iterates as mentioned over the different segments of the paths. This reduces the problem of finding all intersection times into the problem of finding the intersection times for every pair of segments. This is done via bisection of the time intervals, working with certain extended boundingboxes (see below). If the parts[7] `pX` of `p` and `qY` of `q` have extended boundingboxes that intersect, they are

5. The `eps` (not `epsilon`) is not a mistake. It will make our examples work. This sensitivity suggests that the method we use is non-optimal.
6. It is in fact a circular linked list. These are more dynamic in memory, and fits well with the use of control points when paths are cycled.
7. Here you can think of `X` and `Y` as a given sequence of zeros and ones. We add `0` for the subpath corresponding to smaller time values, and `1` for larger.

halved (with respect to time) into pX0, pX1, qY0 and qY1, and the algorithm works on the new parts in the following order:

1. pX0 is tested against qY0,
2. pX0 is tested against qY1,
3. pX1 is tested against qY0,
4. pX1 is tested against qY1.

We give an example to show how the intersectiontimes macro works on a pair of paths consisting of two points each (that is of time length 1, or, if we want, with one segment). We have drawn p and q in Figure 14; the paths intersect at three points. We follow the first few steps of the algorithm in Figures 15–30.

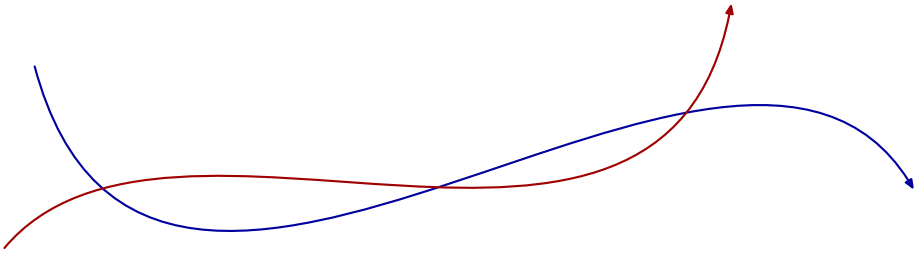


Figure 14.

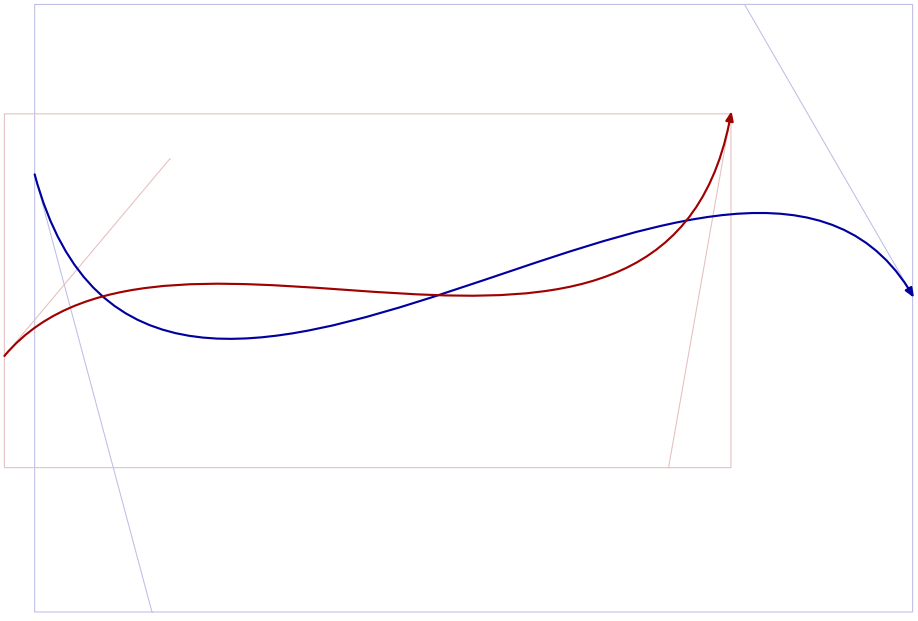


Figure 15. The starting point. For each path we draw its *extended boundingbox*, that is the smallest axis-parallel rectangle that includes both the points and the control points of the path. The extended boundingbox always contains its path, so if the two extended boundinboxes don't intersect[8] then the paths cannot intersect. Here the rectangles intersect. This does not guarantee that the paths intersect; it only means that we should continue with the next step.

8. We say here that two rectangles intersect if they have points (interior, or on the rectangle) in common

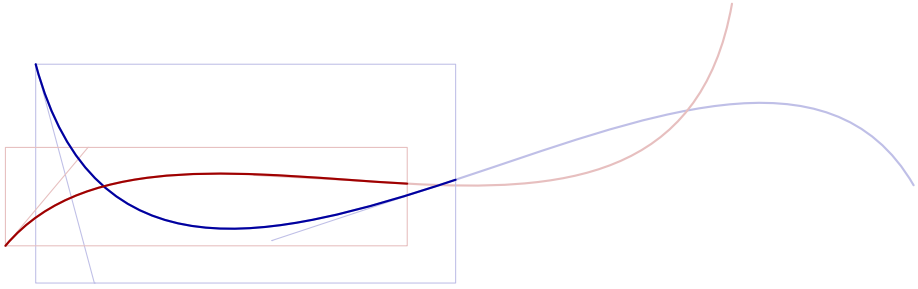


Figure 16. We divide p and q into two pieces at time $1/2$, and call the new paths p0, p1, q0 and q1. The darkblue path is p0 and the darkred one is q1. Since their extended boundingboxes intersect again, we continue with the division.

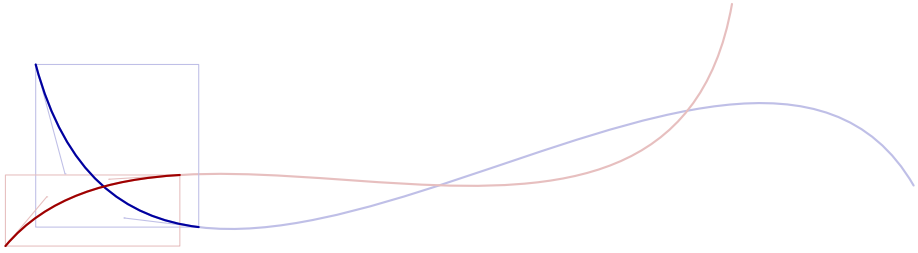


Figure 17. We divide p0 in the middle into p00 and p01. We also divide q0 into two paths q00 and q11. Here we have drawn p00 and q00. Again, we notice that their extended boundingboxes intersect.

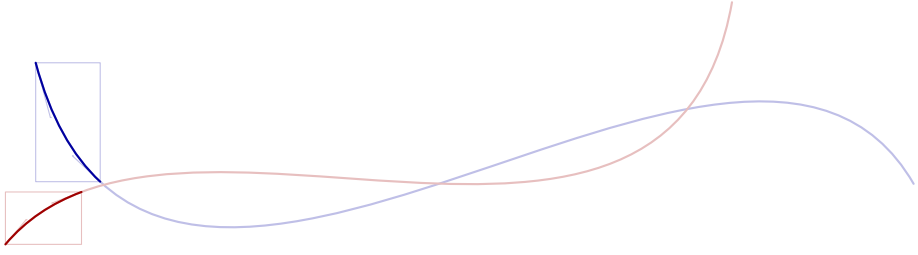


Figure 18. We have divided p00 and q00 in halves. Here we have drawn the paths p000 and q000, and their extended boundingboxes. We note that the boxes do not intersect.

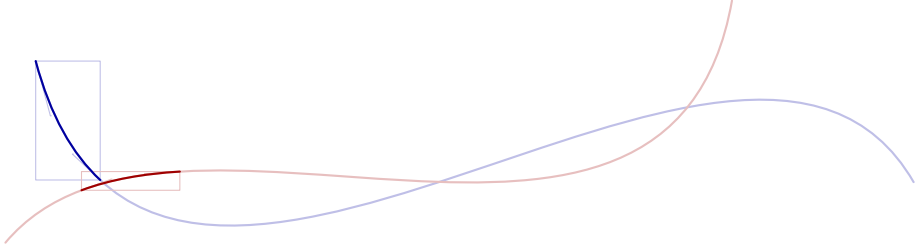


Figure 19. The previous extended boundingboxes did not intersect. We change one of the subpaths, and since we want to have the intersection point as early as possible on the first path p we try first to change the subpath of q. We see that the extended boundingboxes of p000 and q001 do intersect.

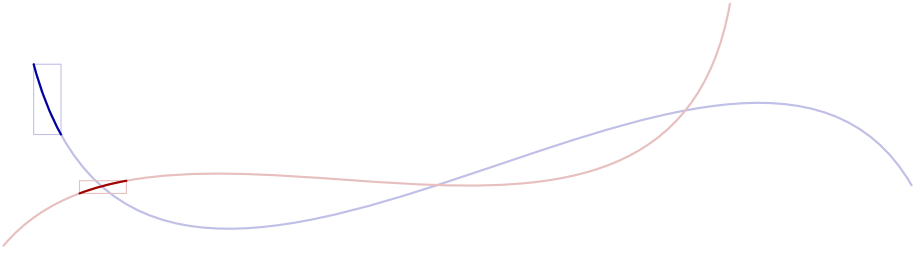


Figure 20. The extended boundingboxes of p0000 and q0010 do not intersect.

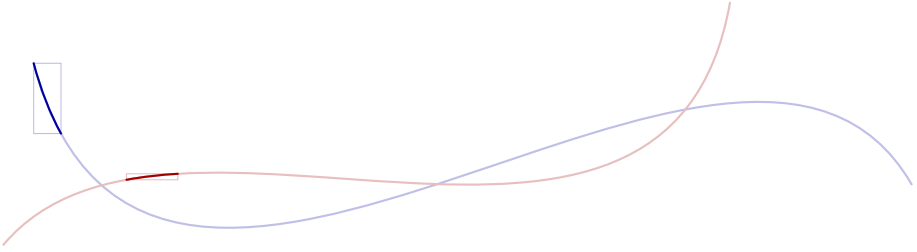


Figure 21. The extended boundingboxes of p0000 and q0011 do not intersect.

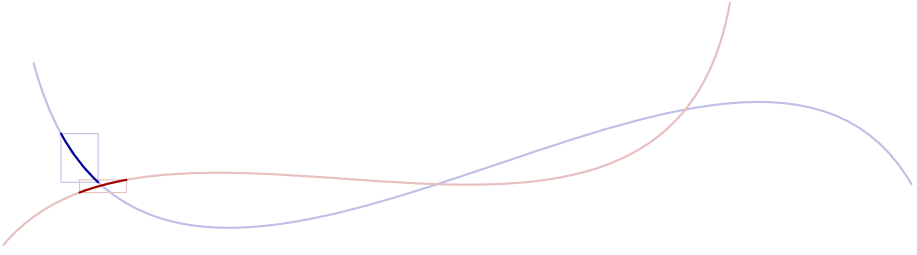


Figure 22. The extended boundingboxes of p0001 and q0010 do intersect.

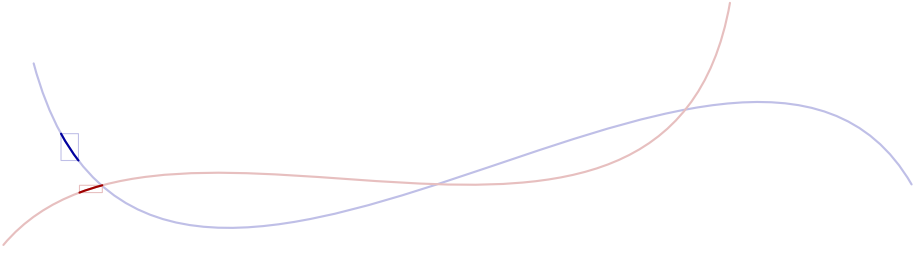


Figure 23. The extended boundingboxes of p00010 and q00100 do not intersect.

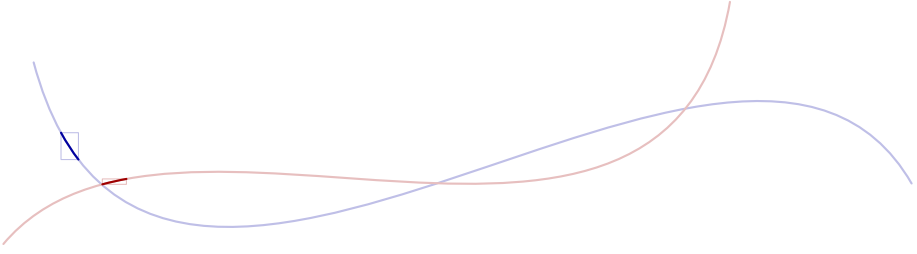


Figure 24. The extended boundingboxes of p00010 and q00101 do not intersect.

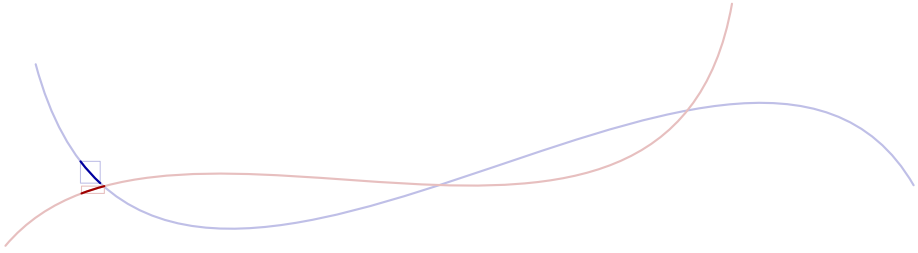


Figure 25. The extended boundingboxes of p00011 and q00100 do not intersect.

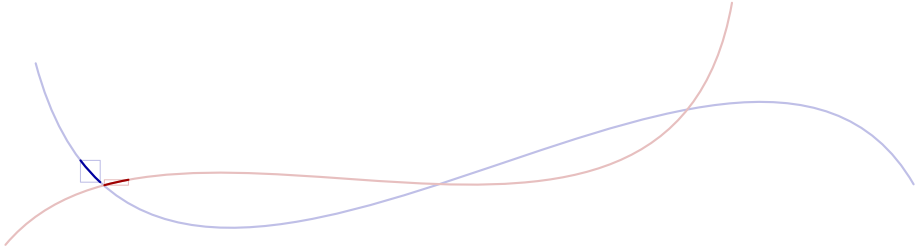


Figure 26. The extended boundingboxes of p00011 and q00101 do not intersect. This means that we did enter a dead-end.

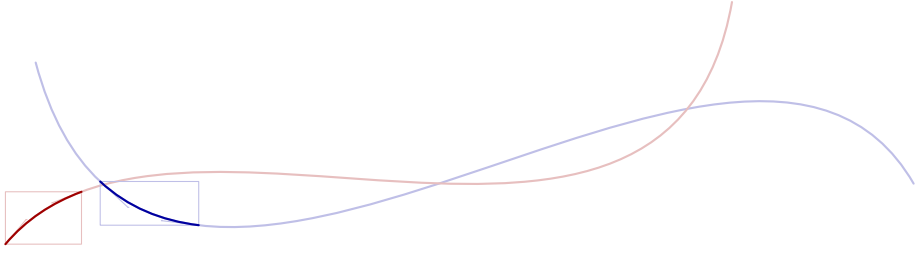


Figure 27. The dead-end pushed us back here. The extended boundingboxes of p001 and q000 do not intersect.

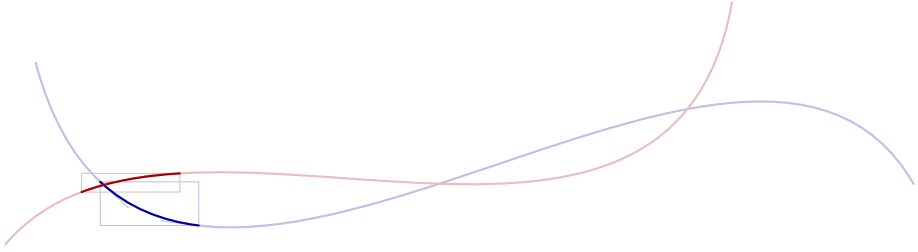


Figure 28. The extended boundingboxes of p001 and q001 do intersect.

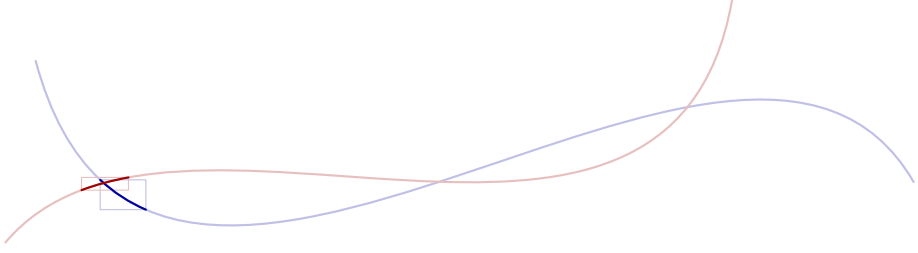


Figure 29. The extended boundingboxes of p0010 and q0010 also intersect.

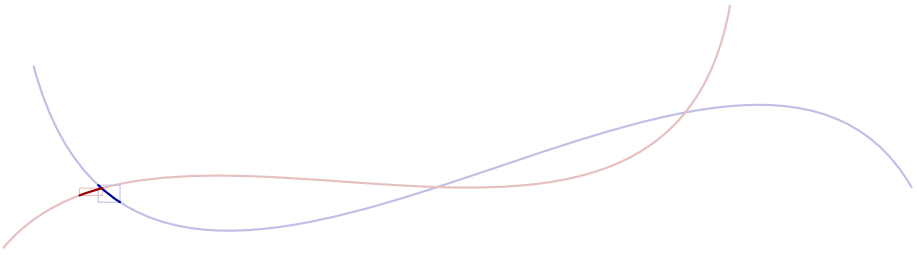


Figure 30. The extended boundingboxes of p00100 and q00100 also intersect.

We could in principle continue this iteration as long as it is numerically meaningful, but we stop here. In Figure 30 we note that the small subpaths actually do intersect. This happens along the path p00100, which means that the time t along p satisfies the inequality

$$\frac{4}{32} = 0.00100_2 \leq t \leq 0.00101_2 = \frac{5}{32}.$$

Hence, if we set t to be the mean value

$$t = \frac{1}{2}\left(\frac{4}{32} + \frac{5}{32}\right) = \frac{9}{64},$$

then the error (in time!) we make is bounded by

$$\frac{1}{2}\left(\frac{5}{32} - \frac{4}{32}\right) = \frac{1}{64}.$$

In practice the `intersectiontimes` macro has some predefined tolerance, and checks if the size of the extended boundingboxes are smaller than that tolerance. Once the sizes are smaller than the tolerance, it considers the paths to intersect, exits the loop and returns a pair with the times along the paths p and q. If it does not find an intersection, it returns the pair `(-1,-1)`.

The final(?) solution

We saw in the previous section how `p intersectiontimes q` works with bisections to find one intersection of p and q. We have seen that often, but not always, it returns the first intersection (measured in time along p).

Hans Hagen realised that instead of exiting the loop when an intersection is found, one could try again with the same pair segments, but this time neglecting the intersection time that was found. In the code in the engine version the crossing is located with a dedicated function that communicates via variables that are global to the instance.

One could perhaps argue that it would be better to cut and run on the different subpaths, or to just continue directly when the intersection was found. But that would introduce other complications and inaccuracies.

Because we suffer from the same issues as the macro approach we described above, there are three extra tricks used: we have an extra counter that makes sure that when we have some deadlock we can get out of it (like deadcycles in TEX). We also need to ignore duplicates that we get because of the small step and inaccuracies; for that we need to trim 8 bits resolution in order be on the safe side. The test examples of the macro variant was instrumental here. Finally we explicitly need to check the end points because otherwise we miss them, which again sounds familiar from the macro variant.

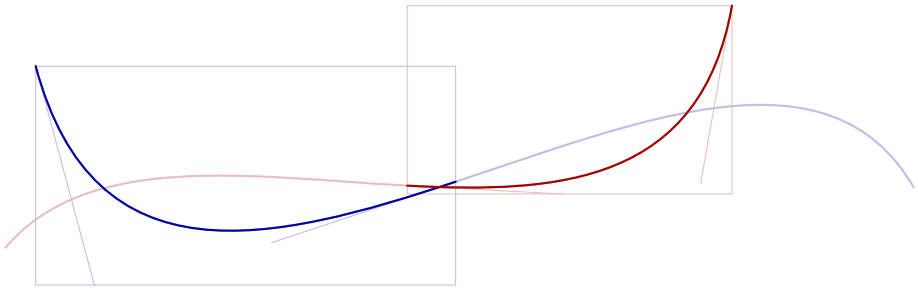


Figure 31. The extended boundingboxes of p0 and q1 do intersect.

In the step-by-step example we have shown above, we only look at the first halves of p and q. All subpaths are followed by a zero, p0X and q0Y. This is what we get in the first run. If we would have continued, we would have got sufficiently small rectangles that meet, and the first intersection time. In the next run, we first end up at the same intersection time, but reject it, and then, eventually, we would end up in the second half of q. The first pair of rectangles will be the ones given in Figure 31. These rectangles intersect, so the loop will continue; in fact it will find the second intersection point here. That would be run two on the segments.

In the third iteration, we will end up at, and skip, the first two intersections. We will then eventually continue by comparing the second half of p with the first half of q. This is seen in Figure 32; the extended bounding boxes do not intersect, so we get no intersection.

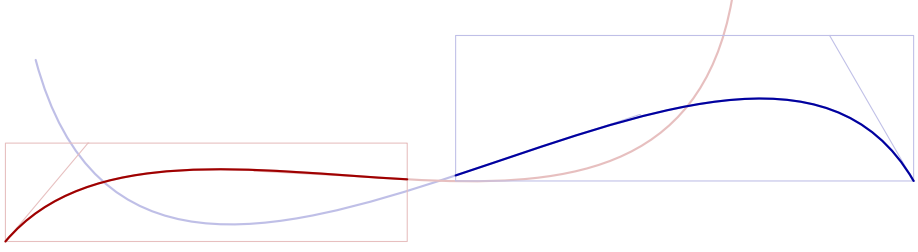


Figure 32. The extended boundingboxes of p1 and q0 do not intersect.

The third iteration continues, and we finally end up in the second half of p and the second half of q, see Figure 33. These intersect, and we will find the third intersection point.

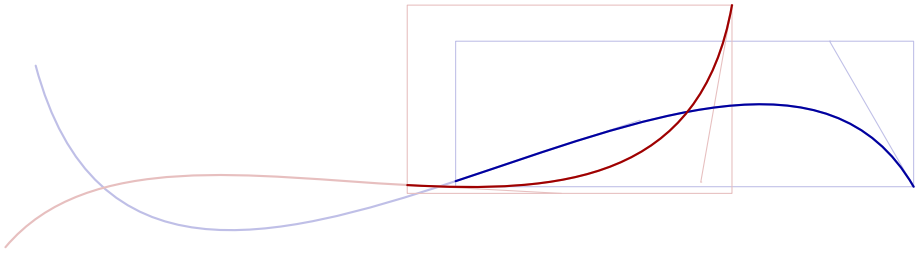


Figure 33. The extended boundingboxes of p1 and q1 do intersect.

When the third intersection point is found, we start the fourth one. This time, we will meet the previous three intersections, and then we will not find any more. The loop is complete. In Figure 34 we have marked the intersections found by the algorithm.

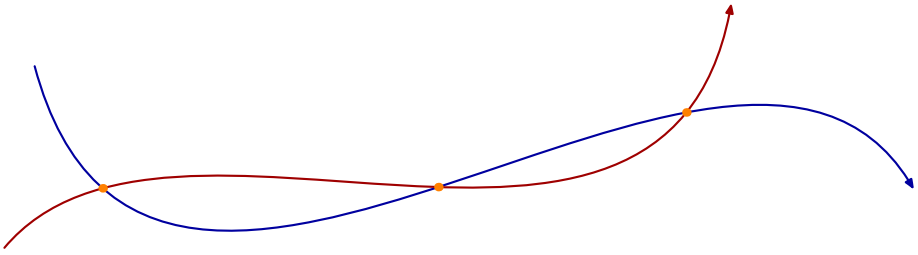


Figure 34.

The new macros

To sum up, we now proudly present a set of new macros. First of all

```
p intersectiontimeslist q
```

This one takes two paths p and q and returns a path with the intersection times, where the first coordinate of each point is the time along p and the second the time along q. Then we have

```
p sortedintersectiontimes q
```

that also returns a path with the intersection times of the paths p and q, sorted in increasing order with respect to the times of p.

To find all intersection points, there is

```
p intersectionpath q
```

that returns a path consisting of the points where p and q intersect, sorted so that the points appear in increasing order with respect to time along p. In fact, this macro first uses `intersectiontimes` to find the times, and then for each pair t of intersection times it returns the middle point of the paths, that is `0.5[xpart t of p,ypart t of q]`. This is the classical approach of `intersectionpoint`. It might not always be what we want, since the points might not be on any of the two paths. To use the points of p or q one can instead use

```
p firstintersectionpath q
```

or

```
p secondintersectionpath q
```

Since the macros `cutbefore` and `cutafter` are relying on `intersectionpoints` we have also introduced

```
p cutbeforefirst q
```

that returns the part of p that remains when the part before the first intersection point with q is cut away, and

```
p cutafterfirst q
```

that returns the part of p that remains when the part after the first intersection point with q is cut away. Similarily, there are

```
p cutbeforelast q
```

and

```
p cutafterlast q
```

that work on the last intersection (along the first path p) of p and q.

Other tools

It is also possible to find all intersection points by using some other common drawing tools. In Figure 35 we show the intersectionpoints of the curves $y = 2\sin(2\pi x)$ and $y = 1/(1 + 2x)$ for $0 \leq x \leq 5$. We say something about each tool below, and provide the code used in each case.

Asymptote

Asymptote is a relative to MetaPost, and it seems that its macro `intersection` uses a similar approach as `intersectiontimes`, inherited from MetaFont. We find the following in Section 6.2 on Paths and guides of the manual [HBP22]:

> "If `p` and `q` have at least one intersection point, return a real array of length 2 containing the times representing the respective path times along `p` and `q`, in the sense of `point(path, real)`, for one such intersection point (as chosen by the algorithm described on page 137 of The MetaFont book). The computations are performed to the absolute error specified by `fuzz`, or if `fuzz < 0`, to machine precision. If the paths do not intersect, return a real array of length 0."

The `intersections` (plural!) macro is used to find all intersection points of two paths. It is not mentioned in the just cited document how it obtains all the intersection points. The code below was placed in a file, and then we ran `asy` on it.

```
unitsize(1cm) ;
import graph ;

real f(real x) { return 2*(sin(2*pi*x)+1) ; }
path A = graph(f,0,5,n=200) ;
draw(A, arrow = Arrow(size = 5), rgb(0, 0, 0.5) + linewidth(1bp)) ;

real g(real x) { return 4/(1+2*x) ; }
path B = graph(g,0,5,n=200) ;
draw(B, rgb(0.5, 0, 0) + linewidth(1bp)) ;

pair vert[] = intersectionpoints(A, B) ;
for(int k = 0 ; k <= vert.length-1; ++k){
  dot(vert[k], orange + linewidth(6bp)) ; }
```

MetaPost/MetaFun

We use the new macro `intersectionpath` to find all intersections between two paths.

```
\startMPcode[instance=doublefun]
path A, B, C ;
A := function(1, "x", "2*sin(2*pi*x) + 2" ,0 ,5 ,1/40) scaled(1cm) ;
B := function(1, "x", "4/(1 + 2*x)" , 0, 5, 1/40) scaled(1cm) ;
C := (A intersectionpath B) ;
drawarrow A withcolor darkblue ;
draw B withcolor darkred ;
drawpoints C withpen pencircle scaled 6bp withcolor "orange" ;
\stopMPcode
```

Pstricks

For pstricks one can use the `pst-intersect` package [Ber14]. It is not clear from the manual how it finds the intersections, but skimming the code, there seems to be some kind of bisection going on, with the clipping of paths (my brain is not constructed to read PostScript code). It is mentioned in the manual that it borrowed the Graham Scal algorithm to calculate the convex hull of a set of points from Bill Casselman's wonderful book *Mathematical Illustrations* (kindly made available online [Cas05]). The implementation in `pst-intersect` can also handle higher order Bézier curves, and

perhaps it is for these that the convex hull algorithm is needed. The LaTeX packages `multi-do`, `pst-intersect` and `xcolor` were loaded in the example below.

```
\begin{pspicture}(5,4.4)
\pssavepath[linecolor=blue!50!black,linewidth=1bp]{A}{
  \psplot[plotpoints=200,arrows=->]{0}{5}{x 360 mul sin 1 add 2 mul} }
\pssavepath[linecolor=red!50!black,linewidth=1bp]{B}{
  \psplot[plotpoints=50]{0}{5}{4 2 x mul 1 add div} }
\psintersect[linecolor=orange,showpoints,linewidth=2bp]{A}{B}
\end{pspicture}
```

Tikz

Tikz has in the `intersections` library support for finding all intersections of two arbitrary paths. It is not clear from the manual [Tan22] how it works, but in Section 13.3.2 we can read

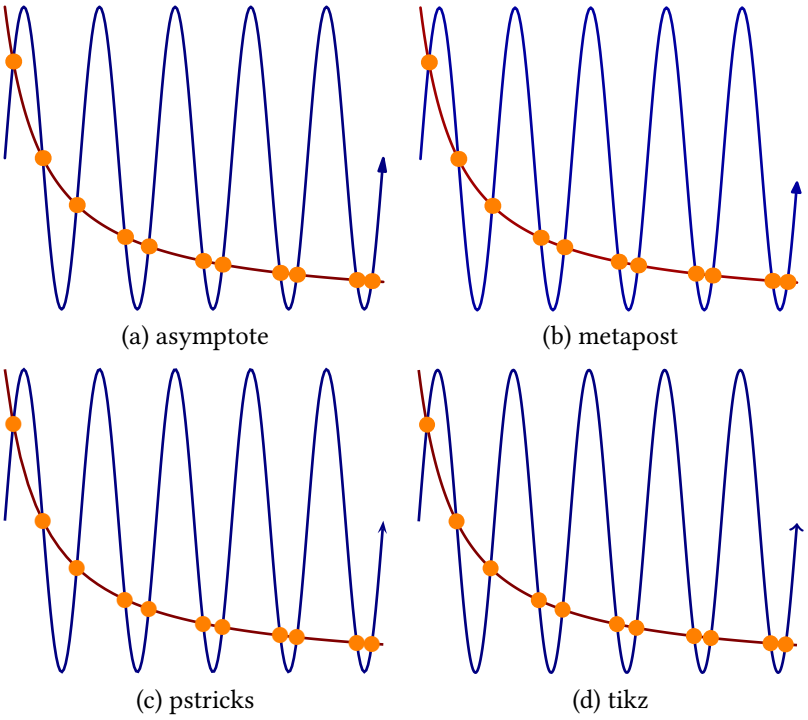
> "This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of TeX, the paths should not be 'too complicated'. In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths."

In the example below we have loaded the `tikz` package, the `intersections` library and also defined two colors

```
\definecolor{darkblue}{rgb}{0, 0, 0.5}
\definecolor{darkred}{rgb}{0.5, 0, 0}
```

Then we used this code.

```
\begin{tikzpicture}
\draw[name path=A, variable=\x, domain=0:5, samples=200,
      line width=1bp, smooth, color=darkblue, ->]
    plot (\x, {2*sin(2*pi*\x r) + 2});
\draw[name path=B, variable=\x, domain=0:5, samples=200,
      line width=1bp, smooth, color=darkred]
    plot (\x, {4/(1+2*\x)});
\fill[name intersections={of=A and B, sort by=line,
      name=i, total=\t}, color=orange]
  \foreach \s in {1,...,\t}{(i-\s) circle (3bp)};
\end{tikzpicture}
```



(a) asymptote (b) metapost

(c) pstricks (d) tikz

Figure 35.

Final words

As we have seen, to find all intersections of two paths is not a trivial task, but now we MetaPost users have reliable engine macros that work. Even with our new macros, however, things can go wrong. Some paths are simply tricky to work with, and sometimes some manual tweaking is required. Let us give one such example.

We try to find the intersections of $y = 4\cos(x) + \cos(2x)$ and $y = 5\cos(x)$, for $0 \leq x \leq 8\pi$. The problem with these curves is that they coincide to order two at the maximas, located at $x = 2\pi k$ for $k \in \{0, 1, 2, 3, 4\}$. This means that the curves are not really crossing there. One could probably find good arguments both for and against that these points should be returned by the algorithm. As the situation is, it returns some, and it depends on how many points we use when constructing the paths. In the code below we use a step size of 0.05 for x. As can be seen in Figure 36, some points are missing. If we change the step size to 0.01 we will see that all the points except the last one is there.

```
\startMPcode
path p, q, b ;
p := function(2, "x", "4*cos(x)+cos(2*x)", epsed(0), epsed(8*pi), 0.05)
  scaled 15 ;
q := function(2, "x", "5*cos(x)", epsed(0), epsed(8*pi), 0.05) scaled 15;
b := (p intersectionpath q) ;

drawarrow p withcolor darkblue;
drawarrow q withcolor darkred;
drawpoints b withcolor "orange" ;
\stopMPcode
```

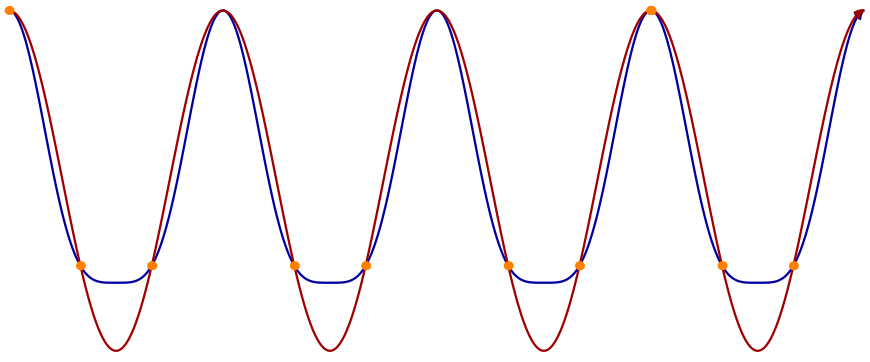
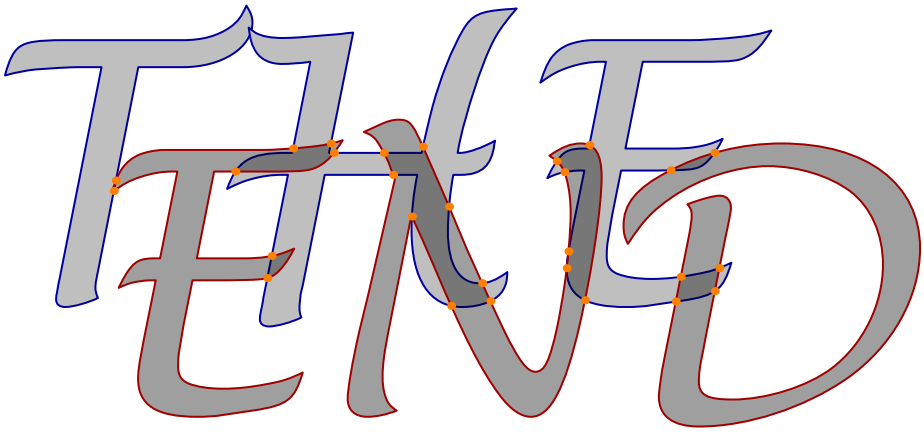


Figure 36.

It would feel wrong to end this article, that is written with a big smile in the face, with a negative example. We give instead an example where all is going well, and why not use the nice feature to extract outlines of characters?

```
\startMPcode
picture p ; p := lmt_outline [ text = "THE" ] scaled 15 ;
picture q ; q := lmt_outline [ text = "END" ] scaled 15 ;
q := q shifted (60, -50) ;

for pp within p :
  fill pathpart pp withcolor 0.75white ;
  draw pathpart pp withpen pencircle scaled 1bp withcolor darkblue ;
endfor ;

for qq within q :
  fill pathpart qq withcolor 0.25white withtransparency (2, 0.5) ;
  draw pathpart qq withpen pencircle scaled 1bp withcolor darkred ;
endfor ;

for pp within p :
  for qq within q :
    path r ; r := (pathpart pp) intersectionpath (pathpart qq) ;
    if known r :
      drawpoints r withpen pencircle scaled 4bp withcolor "orange" ;
    fi ;
  endfor ;
endfor ;
\stopMPcode
```



Acknowledgements

Neither the new engine macros nor this article would have existed if it not were for Hans Hagen, and his open mind to new ideas. Thank you!

References

[HBP22] A. Hammerlindl, J. Bowman, and T. Prince, *Asymptote*, https://asymptote.sourceforge.io/doc/index.html (2022). (version: 2.80-35)

[Cas05] B. Casselman, http://www.math.ubc.ca/ cass/graphics/text/www/ (2005).

[Hen08] T. Henderson, https://tug.org/pipermail/metapost/2008-October/001467.html (2008). (version: October, 2008)

[Hob20] J.D. Hobby, *METAPOST: A users's manual*, https://www.tug.org/docs/metapost/mpman.pdf (2020). (version: 2020)

[Knu86] D. Knuth, *METAFONT: The Program* (Addison Wesley Pub. Co, Reading, Mass, 1986).

[Ber14] C. Bersch, *Pst-intersect: Intersecting arbitrary curves*, https://github.com/cbersch/pst-intersect (2014). (version: March 16, 2014)

[Thu17] T. Thurston, *Drawing with Metapost*, https://github.com/thruston/Drawing-with-Metapost (2017). (version: March 2017)

[Tan22] T. Tantau, *The TikZ and PGF Packages: Manual for Version 3.1.9a*, https://github.com/pgf-tikz/pgf (2022). (version: March 29, 2022)

Mikael P. Sundqvist
mickep@gmail.com

Cyrillisch in publieke fonts

In deze dagen zien we meer cyrillisch om ons heen en het is waarschijnlijk dat met het toetreden tot de EU we dit script wat meer zullen tegenkomen in publicaties. Er zijn destijds wel discussies geweest over het al dan niet toevoegen van grieks en cyrillisch aan bijvoorbeeld de TEXgyre collectie, maar veel is daar niet van terecht gekomen. Een van de redenen was dat er geen goede publieke fonts waren die als startpunt konden dienen, een andere was dat niemand zich opwierp om de kwaliteitscontrole te doen. Zoals veel in de TEX wereld afhangt van vrijwilligers bleek er gewoon geen draagvlak te zijn: een kar moet wel getrokken worden.

Een gevolg is dat in TEX distributies er niet zo heel veel fonts zijn die zowel ‚Latin' (`latn` in OpenType speak) als ‚Cyrillic' (`cyrl`) ondersteunen. Echter, het geval wil dat het font dat we gebruiken voor de Maps het wel ondersteunt. Hieronder laten we vijf fonts zien die men kan gebruiken. De tekst komt uit de Universele Verklaring van de Rechten van de Mens.

Libertinus: Стаття 15.
1. Кожна людина має право на громадянство.
2. Ніхто не може бути безпідставно позбавлений громадянства або права змінити своє громадянство.

Libertinus: Artikel 15
1. Een ieder heeft het recht op een nationaliteit.
2. Aan niemand mag willekeurig zijn nationaliteit worden ontnomen, noch het recht worden ontzegd om van nationaliteit te veranderen.

Libertinus: Article 15
1. Everyone has the right to a nationality.
2. No one shall be arbitrarily deprived of his nationality nor denied the right to change his nationality.

Plex: Стаття 15.
1. Кожна людина має право на громадянство.
2. Ніхто не може бути безпідставно позбавлений громадянства або права змінити своє громадянство.

Plex: Artikel 15
1. Een ieder heeft het recht op een nationaliteit.
2. Aan niemand mag willekeurig zijn nationaliteit worden ontnomen, noch het recht worden ontzegd om van nationaliteit te veranderen.

Plex: Article 15
1. Everyone has the right to a nationality.
2. No one shall be arbitrarily deprived of his nationality nor denied the right to change his nationality.

Dejavu: Стаття 15.
1. Кожна людина має право на громадянство.
2. Ніхто не може бути безпідставно позбавлений громадянства або права змінити своє громадянство.

dejavu: Artikel 15
1. Een ieder heeft het recht op een nationaliteit.
2. Aan niemand mag willekeurig zijn nationaliteit worden ontnomen, noch het recht worden ontzegd om van nationaliteit te veranderen.

Dejavu: Article 15
1. Everyone has the right to a nationality.
2. No one shall be arbitrarily deprived of his nationality nor denied the right to change his nationality.

Xits: Стаття 15.
1. Кожна людина має право на громадянство.
2. Ніхто не може бути безпідставно позбавлений громадянства або права змінити своє громадянство.

Xits: Artikel 15
1. Een ieder heeft het recht op een nationaliteit.
2. Aan niemand mag willekeurig zijn nationaliteit worden ontnomen, noch het recht worden ontzegd om van nationaliteit te veranderen.

Xits: Article 15
1. Everyone has the right to a nationality.
2. No one shall be arbitrarily deprived of his nationality nor denied the right to change his nationality.

Iwona: Стаття 15.
1. Кожна людина має право на громадянство.
2. Ніхто не може бути безпідставно позбавлений громадянства або права змінити своє громадянство.

Iwona: Artikel 15
1. Een ieder heeft het recht op een nationaliteit.
2. Aan niemand mag willekeurig zijn nationaliteit worden ontnomen, noch het recht worden ontzegd om van nationaliteit te veranderen.

Iwona: Article 15
1. Everyone has the right to a nationality.
2. No one shall be arbitrarily deprived of his nationality nor denied the right to change his nationality.

Zoals gezegd, de TEXgyre fonts zouden een uitbreiding kunnen gebruiken, maar dan moet het font team wel worden aangemoedigd en ondersteund. Vanuit de NTG zijn in het verleden substantieel financiële middelen vrijgemaakt voor font projecten en de reserves staan toe dat we een aan een project voor meer scripts in deze fonts bijdragen aan de Gust Font project. Misschien dat de tijd er nu wel rijp voor is. Maar we hebben dan vrijwilligers (lees: gebruikers) nodig die het font team daarin ondersteunen.

Hans Hagen

Y. Robbers

Tante Lenie weet raad...

Uw trouwe steun en toeverlaat voor al uw problemen

Abstract
Deze keer helpt Tante Lenie enkele NTG-leden met hun TEX-problemen en andere diepe zieleroerselen. Zo helpt ze Tamara J., ontwerpster van bordspellen, om mooie dobbelstenen af te beelden in de handleiding voor haar nieuwe spel, met behulp van LATEX, en helpt ze docent klassieke talen Jaap T. om woorden te markeren in een tekst voor een proefwerk dat hij in XƎLATEX maakt. Tenslotte helpt ze Herman R., een wiskundige die vastliep met boldfaced wiskundeformules in sectietitels in plain TEX.

Lieve lezers,

Ieder zichzelf respecterend tijdschrift, krant of zelfs televisieprogramma heeft een oudere dame met wat meer levenservaring dan gemiddeld bij wie mensen terecht kunnen met hun problemen. Ware liefde? Medische problemen? Puistjes? The Guardian en de Daily Mirror hebben Philippa Perry, Tina heeft Djamila, en Oprah heeft Dr. Phil. En vanaf dit nummer heeft MAPS haar eigen Tante Lenie bereid gevonden deze rol te vervullen! Niets is te gek, je hoeft je nergens meer voor te schamen. Of je nu problemen hebt met een obscuur accent in het Tochaars dat je zelfgedefinieе4erde afbreekregels verprutst (slordig in je nieuwste boek dat je met LATEX maakt), of dat je bijdrage voor de MAPS niet compileert in ConTEXt (omdat je per ongeluk `\startarticle` in plaats van `\startArticle` hebt gebruikt, misschien?). Tante Lenie weet raad, en helpt je, desnoods geheel anoniem!

Dobbelsteentjes

Lieve Tante Lenie,

Voor mijn nieuwe bordspel ben ik de handleiding aan het schrijven. In de handleiding wil ik graag dobbelstenen afbeelden. In plaatjes is dat geen probleem, maar hoe doe ik dat in lopende tekst zonder gehannes met plaatjes waardoor m'n regelhoogte verandert? Oh ja, ik gebruik pdfLATEX.

Liefs van
Tamara J.

Lieve Tamara,

Wat leuk! Ik hou erg van bordspellen. Voor jouw probleem zijn reeds diverse oplossingen beschikbaar. Op basis van eps, op basis van TikZ, maar ook gewoon op basis van een dobbelsteen-font, gemaakt in Metafont. De laatste lijkt me voor jou het meest geschikt. In de meeste TEX-distributies is dit font gewoon meegeleverd, maar anders vind je het op CTAN[1]. Dus alles wat je moet doen is het in de preamble van je document een naam geven:

```
\newfont\dice{dice3d}
```

Vervolgens kun je in de lopende tekst dobbelsteentjes opnemen met het commando `\dice`. Bijvoorbeeld:

```
Rol je een {\dice 1}, dan mislukt je aanval sowieso,
bij een {\dice 6} win je sowieso, en bij een
{\dice 2345} mag je tegenstander een dobbelsteen
rollen ter verdediging.
```

Rol je een ⚀ dan mislukt je aanval sowieso, bij een ⚅ win je sowieso, en bij een ⚁⚂⚃⚄ mag je tegenstander een dobbelsteen rollen ter verdediging.
Wil je dobbelsteentje in 3D afbeelden, gebruik dan

```
{\dice 1a 1b 1c 1d 2a 2b 2c 2d 3a 3b 3c 3d
       4a 4b 4c 4d 5a 5b 5c 5d 6a 6b 6c 6d}
```

waarbij het getal bepaalt wat er op de bovenkant van de dobbelsteen staat, en de letter wat er op de voorkant staat (daar zijn natuurlijk maar 4 opties voor, want de bovenkant ligt al vast, en daarmee de onderkant ook: de som is immers altijd zeven. Omdat ook de verdere nummering van dobbelstenen is vastgelegd, is de derde afgebeelde zijde duidelijk zodra je de andere twee hebt bepaald.
Ik hoop dat dit helpt, Tamara. Mocht je behoefte hebben aan andere dobbelstenen, twintigzijdige bijvoorbeeld, dan is daar nog geen pasklare oplossing voor, maar wellicht is dit een leuke uitdaging voor een van onze lezertjes?

Liefs,
je tante Lenie.

Sleutelwoorden onderstrepen

Beste Tante Lenie,

Ik ben een docent klassieke talen aan een bekend Gymnasium. Alweer enkele jaren geleden heeft een collega Biologie mij laten kennismaken met XeLaTeX en sindsdien wil ik niets anders meer! Het is zo gemakkelijk om teksten in het oud-Grieks te maken en ze er prachtig uit te laten zien! Alleen, als ik een proefwerk Latijn of Grieks maak, staat daar vaak een klein aantal voor mijn leerlingen nieuwe woorden in. Die onderstrepen we altijd (ja, ik weet het: dat is lelijk, maar tradities zijn nu eenmaal heilig bij ons classici), zodat leerlingen weten dat ze die in het bijgevoegde woordenlijstje kunnen vinden. Alleen soms onderstreep ik zo'n woord wel een paar keer, maar niet elke keer. Is daar een oplossing voor? Kan ik die woorden bijvoorbeeld automatisch laten onderstrepen? M'n collega Wouter die Engels geeft wist geen oplossing, maar zou dit zelf ook graag kunnen!

Groetjes,
Jaap T.

Lieve Jaap,

Ik snap je probleem... één zo'n woord zie je nu eenmaal gemakkelijk over het hoofd in een tekst. Gelukkig is TeX gemaakt voor het manipuleren van tekst, en zou dit dus eenvoudig te automatiseren moeten zijn. En je hoeft het niet eens zelf te doen, er is al een pakketje voor geschreven! Het heet `xesearch`[2], werkt met elke taal die je maar wilt — dus ook polutoniko-grieks — en heeft een duidelijke handleiding. Het kan echter veel meer dan jij wilt, dus ik zal de essentie even voor je samenvatten. In de preamble van je document laad je het pakketje in, en definieer je de lijst met woorden waar het om gaat. Die lijst moet een naam hebben, en elke woord dat je wilt markeren zet je erin. Je definieert ook wat er met dat woord moet gebeuren, in jouw geval onderstrepen.

```
\usepackage{xesearch}
\SearchList{woordenlijst}{underline{#1}}
           {serpens,hortus,horto}
```

En vervolgens gaat de rest vanzelf: Serpens in horto. Marcus et Cornelia in horto ambulant. Etcetera, etcetera. Je ziet dat de hoofdletters automatisch ook herkend worden, maar de verschillende naamvallen moet je natuurlijk zelf wel even elk apart in de lijst zetten. Je kunt het overigens net zo gemakkelijk inladen in LaTeX (met dezelfde syntax), plain TeX of ConTeXt.

Voor plain TeX:

```
\input xesearch.sty
```

En voor ConTeXt:

```
\usemodule[xesearch]
```

En dit werkt natuurlijk in alle ondersteunde talen, dus ook in het oud-Grieks!

liefs,
je tante Lenie.

Wiskunsten

Lieve Tante Lenie,

Voor mijn proefschrift over oneindigdimensionale elliptische curven met complexe parameters wil ik graag formules opnemen in sommige hoofdstuktitels. Die hoofdstuktitels gebruiken een bold font, maar m'n formules blijven in roman verschijnen, wat ik ook doe. Kunt u mij helpen? Oh ja, ik gebruik plain TeX en de Computer Modern fonts.

Liefs,
Herman R.

Lieve Herman,

Natuurlijk kan ik je helpen! Belangrijk is dat je eerst even de boldmath fonts definieert. Dat kan bijvoorbeeld zo:

```
\font\tenib=cmmib10
\font\sevenib=cmmib7
\font\fiveib=cmmib5
\font\tensyb=cmbsy10
\font\sevensyb=cmbsy7
\font\fivesyb=cmbsy5
\font\tenexb=cmexb10
```

en dan vervolgens zorgen we dat we deze fonts met het `\boldmath`-commando gemakkelijk kunnen gebruiken.

```
\def\boldmath{%
  \textfont0=\tenbf
  \textfont1=\tenib
  \textfont2=\tensyb
  \textfont3=\tenexb
  \scriptfont0=\sevenbf
  \scriptfont1=\sevenib
  \scriptfont2=\sevensyb
  \scriptfont3=\tenexb
  \scriptscriptfont0=\fivebf
  \scriptscriptfont1=\fiveib
  \scriptscriptfont2=\fivesyb
  \scriptscriptfont3=\tenexb }
```

En het enige dat je dan nog moet doen, is je sectiebegin op de juiste manier definiëren om ook gebruik te maken van de boldmath fonts:

```
\catcode`@=11
\outer\def\beginsection#1\par{\vskip\z@
  plus.3\vsize\penalty-250
  \vskip\z@ plus-.3\vsize\bigskip\vskip\parskip
  \message{#1}\leftline{\bf\boldmath#1}
  \nobreak\smallskip\noindent}
\catcode`@=12
```

En kun je aan de slag! Even een voorbeeldje:

```
\beginsection
  De formule $\oint_{-1}^1 {\sin x \over x} dx$

De formule $\oint_{-1}^1 {\sin x \over x} dx$
```

En dat ziet er dan zo uit:

De formule $\boldsymbol{\oint_{-1}^{1} \frac{\sin x}{x} dx}$

De formule $\oint_{-1}^{1} \frac{\sin x}{x} dx$

Hopelijk lost dit je probleem op, Herman!

Liefs,
je tante Lenie

Slotwoord

En dat was het weer voor deze keer! Heb je zelf een vraag? Stuur hem aan de MAPS-redactie, en zij zullen hem aan mij doorsturen. Pas goed op jezelf en TEX vrolijk verder!

Liefs,
jullie tante Lenie

Note

Footnotes

1. https://ctan.org/pkg/dice
2. https://ctan.org/pkg/xesearch

Yuri Robbers

Dice3D OpenType

(quick font hack two)

Abstract
The previous article by Yuri Robbers shows simulated 3D dice. That font existed only as a MetaFont source file, so for the ConTEXt-format Maps article, I had to quickly create an OpenType version.

Using MetaPost instead of MetaFont

The font comes as a single file called `dice3d.mf`, which contains all needed definitions. It is intended for the `plain.mf` macros with no extra macro packages needed. This makes it possible to run the file directly in MetaPost, which a slightly complicated command line:

```
mpost --mem=mfplain
  -s outputtemplate='"%j-%c.eps"'
  '\\mag=36; mode=lowres; input dice3d.mf'
```

the bits and pieces:

`--mem=mfplain`
: This preloads the `mfplain.mp` file, which mimics the MetaFont `plain` macros.

`-s outputtemplate='"%j-%c.eps"'`
: This make MetaPost produce eps files with nice names like `dice3d-99.eps` instead of `dice3d.99`.

`'\mag=36; mode=lowres; input dice3d.mf'`
: For font-making, it is necessary to set up the MetaFont 'mode' and magnification. This magnification setting ensures that the output images are such that they fit the typical 1000-units-per-em for a PostScript-type font. The mode is a predefined mode in `mfplain` that distorts the image as little as possible while still being a font-making mode (as opposed to 'proof' mode).

  This part of the command line is enclosed in quotes and starts with a backslash so that we can set these parameters without having to alter the actual font source. The backslash prevents the assumed `input` command at the start of the line. The internal `jobname` is then set by the next actual `input` command.

After running the command, there are 30 eps files.

Here is one as an example:

A bit of FontForge

Those images are then imported into FontForge in their respective slots. Once all 30 are imported, they are tweaked a little as usual:

- ☐ All the points are rounded to integers
- ☐ All overlap is removed
- ☐ All right side-bearings are set to be equal to the left side-bearings.

All those actions are single commands after selecting all font glyphs, which makes this process really fast.

Then, as this font as some ligatures, a `liga` table needs to be created. I did this manually for this font because it is so simple (the ligature table is listed at the end of `dice3d.mf`):

```
%   ligature tables for 3D dice:
%   #a, #b, #c, #d, where # is the value on the top face,
%                   and the letter indicates the value on
%                   the front face: "a" -> smallest,
%                                   "d" -> largest
ligtable "1": "a" =: "a", "b" =: "b", "c" =: "c", "d" =: "d";
ligtable "2": "a" =: "e", "b" =: "f", "c" =: "g", "d" =: "h";
ligtable "3": "a" =: "i", "b" =: "j", "c" =: "k", "d" =: "l";
ligtable "4": "a" =: "m", "b" =: "n", "c" =: "o", "d" =: "p";
ligtable "5": "a" =: "q", "b" =: "r", "c" =: "s", "d" =: "t";
ligtable "6": "a" =: "u", "b" =: "v", "c" =: "w", "d" =: "x";
```

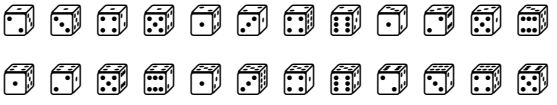
Finally, I adjusted the font name and generated an OpenType version.

Usage

Using the OpenType version is quite straightforward:

```
\font\contextdice=dice3d*default
{\contextdice 1 2 3 4 5 6 \crlf
1a 1b 1c 1d 2a 2b 2c 2d 3a 3b 3c 3d \crlf
4a 4b 4c 4d 5a 5b 5c 5d 6a 6b 6c 6d}
```

⚀ ⚁ ⚂ ⚃ ⚄ ⚅



Taco Hoekwater

The art of Maps proofreading

When Taco had to turn Yuri's article into the Maps format the problem was that the tutorial about dice was using a MetaFont and as the Maps prefers outlines it made sense to go that way. So, as follow up Taco made the three dimensional dice into a font and wondered if we should add a typescript to the distribution. And who can deny Taco. especially when also Frans wants to use these shapes. While pondering this I realized that we already had dice in ConTEXt so I took a look at the MetaFont code to see what magic was needed for going 3D. The file used by Yuri and Taco is `DICE3D.MF` by Thomas A. Heim and dates from 1998.

The OpenType font that Taco made shows some inaccuracies with respect to the way the three sides are connected. At first I though that there was some issue with the transform but it more looks like it is in the definitions of the paths. I won't go into details but using scaled `fullsquare`'s works well so that is what we do here.

In the code below we start with the simple dice shapes. We just define the six variants as macros. Because we later will reuse the dots we save them in a picture list. The definitions have been simplified a bit because in the MetaFun module we also define the reversed variants two and three as well as dominos.

```
\startMPcalculation{simplefun}
  picture DiceDots[] ;

  pickup pencircle scaled 3/2 ;

  DiceDots[ 1] := image ( draw(4,4) ; ) ;
  DiceDots[ 2] := image ( draw(2,6) ; draw(6,2) ; ) ;
  DiceDots[ 3] := image ( draw(2,6) ; draw(4,4) ; draw(6,2) ; ) ;
  DiceDots[ 4] := image ( draw(2,6) ; draw(6,6) ; draw(2,2) ; draw(6,2) ; ) ;
  DiceDots[ 5] := image ( draw(2,6) ; draw(6,6) ; draw(4,4) ; draw(2,2) ; draw(6,2) ; ) ;
  DiceDots[ 6] := image ( draw(2,6) ; draw(6,6) ; draw(2,4) ; draw(6,4) ; draw(2,2) ;
                                                                          draw(6,2) ; ) ;

  def DiceFrame =
    pickup pencircle scaled 1/2 ;
    draw unitsquare scaled 8 ;
  enddef ;

  vardef DiceOne   = DiceFrame ; draw DiceDots[1] ; enddef ;
  vardef DiceTwo   = DiceFrame ; draw DiceDots[2] ; enddef ;
  vardef DiceThree = DiceFrame ; draw DiceDots[3] ; enddef ;
  vardef DiceFour  = DiceFrame ; draw DiceDots[4] ; enddef ;
  vardef DiceFive  = DiceFrame ; draw DiceDots[5] ; enddef ;
  vardef DiceSix   = DiceFrame ; draw DiceDots[6] ; enddef ;

  vardef DiceBad =
    DiceFrame ; draw (1,7) -- (7,1) ; draw (1,1) -- (7,7) ;
  enddef ;
\stopMPcalculation
```

Next we define a Type3 font. The `lmt_` prefix is used for a collection of macros in LuaMetaFun. It is an example of how we enhance the user interface with parsers written in Lua; these sort of extend the MetaPost syntax. These glyphs all have the

same dimensions. A Type3 font consists of bitmap or outline drawing operators and with some Lua magic we can create these in LuaTEX and LuaMetaTEX. One just has to make sure to plug them into the pdf backend.

```
\startMPcalculation{simplefun}

  lmt_registerglyphs [
    name     = "dice",
    units    = 12,
    width    = 8,
    height   = 8,
    depth    = 0,
    usecolor = true,
  ] ;

  lmt_registerglyph [ category = "dice", unicode = "0x2680", code = "DiceOne;" ] ;
  lmt_registerglyph [ category = "dice", unicode = "0x2681", code = "DiceTwo;" ] ;
  lmt_registerglyph [ category = "dice", unicode = "0x2682", code = "DiceThree;" ] ;
  lmt_registerglyph [ category = "dice", unicode = "0x2683", code = "DiceFour;" ] ;
  lmt_registerglyph [ category = "dice", unicode = "0x2684", code = "DiceFive;" ] ;
  lmt_registerglyph [ category = "dice", unicode = "0x2685", code = "DiceSix;" ] ;

  lmt_registerglyph [ category = "dice", private = "invaliddice", code = "DiceBad;" ] ;

\stopMPcalculation
```

We now will add the three dimensional variants for which we need a few transformations that we borrow from the MetaFont original. It is the only code we had to take but it is also the most magical.

```
\startMPcalculation{simplefun}
  transform t[] ; numeric r ; r := sqrt(1/4) ;

  hide((0,0) transformed t1 = (0,0)) ;
  hide((1,0) transformed t1 = (r,r)) ;
  hide((0,1) transformed t1 = (0,1)) ;

  hide((0,0) transformed t2 = (0,0)) ;
  hide((1,0) transformed t2 = (1,0)) ;
  hide((0,1) transformed t2 = (r,r)) ;

  t3 := t1 shifted (8,0) ; % front to right side
  t4 := t2 shifted (0,8) ; % front to top
\stopMPcalculation
```

Next we define the extra variants. The list of combinations (of three digits) come from the mentioned MetaFont file:

```
\startMPcalculation{simplefun}
  vardef Diced(expr a, b, c) =
    draw image (
      pickup pencircle scaled 1/2 ;
      draw image (
        nodraw unitsquare scaled 8 transformed t4  ;
        nodraw unitsquare scaled 8 transformed t3  ;
        nodraw unitsquare scaled 8 ;
        dodraw unitsquare scaled 8 ;
      ) ;
      draw DiceDots[a]                ;
      draw DiceDots[b] transformed t3 ;
      draw DiceDots[c] transformed t4 ;
```

```
    ) ;
  enddef ;
\stopMPcalculation
```

We use this macro when we register the shapes. The Unicode's are of course wrong but we don't care too much about them here. We could have used private slots.

```
\startMPcalculation{simplefun}
  lmt_registerglyph [ category = "dice", unicode = "123", code = "Diced(1,2,3);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "135", code = "Diced(1,3,5);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "142", code = "Diced(1,4,2);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "154", code = "Diced(1,5,4);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "214", code = "Diced(2,1,4);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "231", code = "Diced(2,3,1);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "246", code = "Diced(2,4,6);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "263", code = "Diced(2,6,3);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "312", code = "Diced(3,1,2);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "326", code = "Diced(3,2,6);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "351", code = "Diced(3,5,1);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "365", code = "Diced(3,6,5);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "415", code = "Diced(4,1,5);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "421", code = "Diced(4,2,1);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "456", code = "Diced(4,5,6);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "462", code = "Diced(4,6,2);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "513", code = "Diced(5,1,3);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "536", code = "Diced(5,3,6);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "541", code = "Diced(5,4,1);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "564", code = "Diced(5,6,4);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "624", code = "Diced(6,2,4);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "632", code = "Diced(6,3,2);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "645", code = "Diced(6,4,5);", width = 12, height = 12 ] ;
  lmt_registerglyph [ category = "dice", unicode = "653", code = "Diced(6,5,3);", width = 12, height = 12 ] ;
\stopMPcalculation
```

At the ConTEXt (read: TEX) end we define three font features. The digits features will map digits onto dice and the ligatures, when enabled, come from sequences of thee digits. The `metapost` feature pushes the graphics into the font instead of already present characters.

```
\definefontfeature
  [dice:normal] % no reverse in this example
  [default]
  [metapost={category=dice}]

\definefontfeature
  [dice:digits]
  [dice:digits=yes]

\definefontfeature
  [dice:three]
  [dice:three=yes]
```

The `dice:digits` and `dice:three` features are implemented as follows. The ligature definitions have to come before the digit remapping because we process features in order.

```
\startluacode
  fonts.handlers.otf.addfeature("dice:three", {
    type      = "ligature",
```

```
    order     = { "dice:three" },
    nocheck   = true,
    data      = {
      [123] = { 0x31, 0x32, 0x33 }, [135] = { 0x31, 0x33, 0x35 },
      [142] = { 0x31, 0x34, 0x32 }, [154] = { 0x31, 0x35, 0x34 },
      [214] = { 0x32, 0x31, 0x34 }, [231] = { 0x32, 0x33, 0x31 },
      [246] = { 0x32, 0x34, 0x36 }, [263] = { 0x32, 0x36, 0x33 },
      [312] = { 0x33, 0x31, 0x32 }, [326] = { 0x33, 0x32, 0x36 },
      [351] = { 0x33, 0x35, 0x31 }, [365] = { 0x33, 0x36, 0x35 },
      [415] = { 0x34, 0x31, 0x35 }, [421] = { 0x34, 0x32, 0x31 },
      [456] = { 0x34, 0x35, 0x36 }, [462] = { 0x34, 0x36, 0x32 },
      [513] = { 0x35, 0x31, 0x33 }, [536] = { 0x35, 0x33, 0x36 },
      [541] = { 0x35, 0x34, 0x31 }, [564] = { 0x35, 0x36, 0x34 },
      [624] = { 0x36, 0x32, 0x34 }, [632] = { 0x36, 0x33, 0x32 },
      [645] = { 0x36, 0x34, 0x35 }, [653] = { 0x36, 0x35, 0x33 },
    }
  } )

  fonts.handlers.otf.addfeature("dice:digits", {
    type      = "substitution",
    order     = { "dice:digits" },
    nocheck   = true,
    data      = {
      [0x30] = "invaliddice",
      [0x31] = 0x2680, [0x32] = 0x2681, [0x33] = 0x2682,
      [0x34] = 0x2683, [0x35] = 0x2684, [0x36] = 0x2685,
      [0x37] = "invaliddice",
      [0x38] = "invaliddice",
      [0x39] = "invaliddice",
    },
  } )
\stopluacode
```

We now can use these fonts so we define a few font instances with different features:

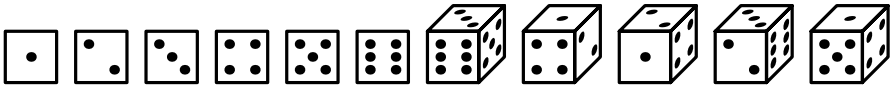
```
\definefont[DiceN][Serif*dice:normal]
\definefont[DiceD][Serif*dice:normal,dice:digits]
\definefont[DiceT][Serif*dice:normal,dice:three,dice:digits]
```

The next few lines give simple two dimensional dice:

```
\DiceN \dostepwiserecurse{"2680}{"2685}{1}{\char#1\quad}%
\DiceD 2\quad5\quad3\quad0
```

As we can see in the next character run, a nice aspect is that the digits are also transformed so there is some perspective in it.

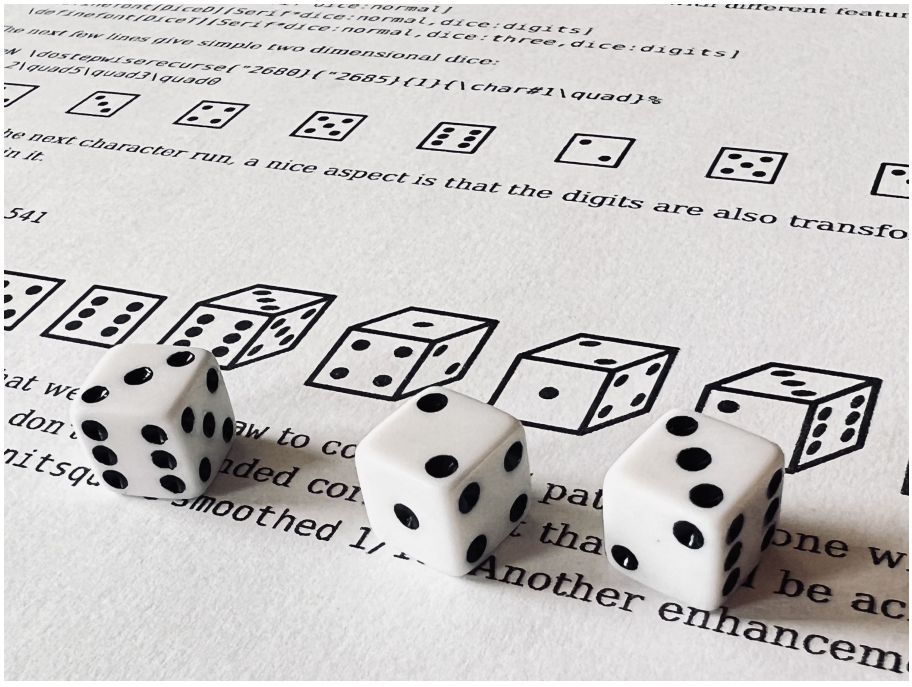
```
\DiceT 1 2 3 4 5 6
\DiceT 653 421 142 263 541
```



One of the probably unseen details is that we use `nodraw` to combine all paths into one which gives nicer shapes. Contrary to the original we don't use rounded corners but that could be achieved by replacing the `unitsquares` by for instance `unitsquare smoothed 1/10`. Another enhancement could be filled dice with white dots.

We come now to the title of this article. When a Maps is being composed, the workflow is as follows. An author send an article, and when it's in pdf format, Frans will read it carefully and feedback issues. When all is okay, Taco kicks in and turns it into the Maps format which involves looking at page breaks, scaling of images, checking fonts and color etc. Then Frans again takes a look at it. It cannot be denied that in my personal case I actually rely on Frans to prevent me from mistakes.

How serious the Maps proofreading is done is demonstrated in the next image. It shows us that not only an article gets printed but that there is some handy work involved too. In this case Frans took real dice as reference. It shows how I get bitten by not showing the complete implementation here.



These show the mirrored variants of two and three and if you really want a consistent set of dice you need to have some use these variants. I will not do that here because we then also have to discuss influencing the variants. It makes sense to use Unicode modifiers to control this. The dice definition in the ConTEXt module actually also have these mirrored shapes and a `dice:reverse` feature:

```
\startMPcalculation{simplefun}
  DiceDots[-2] := image ( draw(6,6) ; draw(2,2) ; ) ;
  DiceDots[-3] := image ( draw(6,6) ; draw(4,4) ; draw(2,2) ; ) ;
\stopMPcalculation
```

But first I have to find me some dice or borrow Frans his reference set. We leave that for the ConTEXt meeting later this year.

Hans Hagen

MetaFun for generative art

Abstract
This article shows how MetaFun can be used to create generative art, by showing the construction of three projects, step by step.

Keywords
MetaFun, art, creation

Introduction

The idea of making generative art with MetaFun was for me a convolution of several elements. The first is a painting by Niele Toroni seen at the Museum of Modern Art of New-York (MOMA) showing imperfect red squares on a canvas: it was incredible. I recently redid a version with circles (not to copy the original version) shown in figure 1. The second element is that I used to make regular representations of random processes in Tikz, to illustrate lecture notes on stochastic processes. Not only these representations are useful, but sometimes beautiful too ! The third and last element is probably the beautiful covers of the ConTEXt manuals, combined with the discovery of the MetaFun manual.

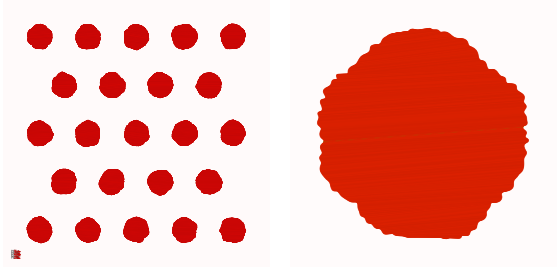


Figure 1. Left a version of Niele Toroni piece with circles instead of squares. Right a zoom on the first circle.

It seemed so easy and beautiful to represent random representations in MetaFun that I explored it. After making many drawings, simple at first, and then more advanced, I discovered that this activity had a name: this art form is called *generative art*. We can borrow the definition from Galenter (2013):

"*Generative art refers to any art practice where the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, which is set into motion with some degree of autonomy contributing to or resulting in a completed work of art.*"

The key for generative art made by a drawing language, is to have random components. Usually, in my drawings, every part, even a tiny one is randomized: position, form, color, etc. An example of this is shown in the right part of figure 1, presenting a small part of the whole piece. As you can see, instead of filling the circle with a unique color, dozens of lines are used trying to give the feeling a real brush, using a randomized color. The accumulation of these details bring organicity to the piece.

So a such a drawing is nothing more than an algorithm with some random components. That means that the same algorithm will produce different results. In general, the more you leave room for randomness, the less predictable the result will be, the more surprising and interesting it will be, but the longer it will take to sort out the successful results among the ten or a hundred drawings made by the same algorithm. So more randomness equals more curation.

It is interesting to note that MetaFun can also be used to make movies (animations) with the help of ConTEXt : a loop in ConTEXt permits to generate easily several hundreds of pages, where some parameters of a drawing change from one page to the next. We can combine these frames, at a rate for example of 30 frames by second, to make a movie, where we can of course add music. Finally, before drawing, let's add that most generative artists use a javascript library named *p5.js*.

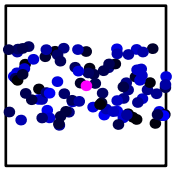
The *randomized* operator

The MetaFun manual is *the* document to read to learn properly the language, and I assume the reader has a familiarity with the basics of the language, although explanations will be as detailed as possible. I emphasize here a particular aspect of MetaFun, the randomized operator, heavily used for generative art. Briefly, this operator adds randomness to almost everything, and this randomness is the key for generative art.

Let's draw a square of size 60, it will be centered at position $(0,0)$, and a point at position $(0,0)$ in color

magenta; then we will draw 50 randomized points around the $(0,0)$ point, in blue, randomized (60,30) :

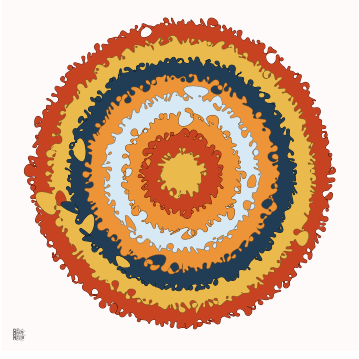
```
draw fullsquare scaled 60 ;
for i=1 upto 50:
       draw (0,0) randomized (60,30)
         withcolor blue randomized(0.1,2) ;
endfor;
draw (0,0) withcolor magenta ;
```



As you can see, the pair of numeric values following the `randomized` operator gives the amount of randomness in both directions, express in regular unit; moreover the color can be randomized too. In fact, as described in the MetaFun manual, "randomized can handle a numeric, pair, path and color, and its specification can be a numeric, pair or color, depending on what we're dealing with."

Project one : delirious circles

MetaFun (MetaPost) is a vectorial language; it is then really easy and natural to draw smooth lines and curves, but it is a little more work to draw *agitated* (i.e. not smooth, chaotic, erratic...) paths. Of course, this is possible, because so many things are possible with MetaFun ! So we begin by building a simple piece illustrating *agitated* paths. Once this first piece is built, we will see how we can use this work to create a more complex piece. Our first objective is to build a piece similar to this :



The first thing to realize this piece is to be able to draw these *agitated* paths. This algorithm is now described and illustrated in figure 2, using a simple circle. In order to *agitate* a path P, the first step is to take a number of points n_1 along P (here $n_1 = 8$) and randomize these points by a quantity t_1 (t for turbulence); then build a path with these n_1 points, and we obtain a new path illustrated as "Step 1" in the illustration. We then repeat this strategy recursively a number of times S (for Steps), using the result of the previous step as the path to be randomized in the current step, taking more points and less noise at each step; this way, the first steps give the global form of the resulting path, and the last steps add some little noise along the path. Varying these parameters, we will obtain different results. Please note that the following MetaFun code is made to be comprehensive more than computationally optimal; let's do it:

```
numeric n[] , t[]; path P;
% Initial values ----------------- ;
n[1] := 8 ; t[1] := 10 ;
% Calculations of t_s and n_s ---- ;
for i=2 upto 5:
  n[i] := n[i-1] * 2 ;
  t[i] := t[i-1] * .8 ;
endfor;
% Initial shape ------------------ ;
P := fullcircle scaled 40;
% Let's add turbulence in S=5 steps ---- ;
for s=1 upto 5 :
 P := for i=1 upto n[s]:
  point (i/(n[s])) along P randomized t[s] ..
    endfor cycle ;
  draw P  ;
  drawpoints P withpen pencircle scaled 2
    withcolor red;
endfor;
```

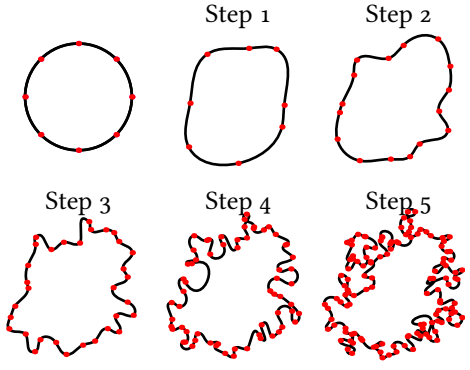


Figure 2. Illustration of the algorithm to *agitate* a path.

We can wrap this in a macro; to keep this simple here, we will assume that the path is a cycle (i.e a closed path), and that the number of points and the noise level at each step are given respectively by

$$n_s = n_0 \times f_n^s, \quad t_s = t_0 \times f_t^s, \quad \text{for } s \geq 1, n_0\, t_0 \text{ known.}$$

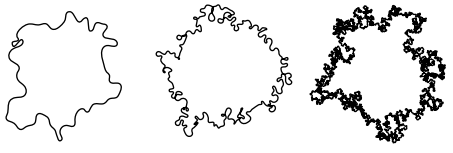
but of course, the macro can be modified for non cycled paths, and others expressions for n_s are t_s are possible. Here is our macro taking a path and others parameters as input and returning an *agitated* path (R):

```
vardef agitate(expr thepath, S, n, fn, t, ft) =
 save R , nbpoints , noiselevel ;
 path R ; nbpoints := n ; noiselevel := t ;
 R := thepath ;
  for s=0 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 ;
```

Please note that the variables `S` and `f_n` need to be reasonable... The processing time is exponential regarding these values, so caution is necessary in experimenting ! Some examples of realizations (starting with a circle again) :

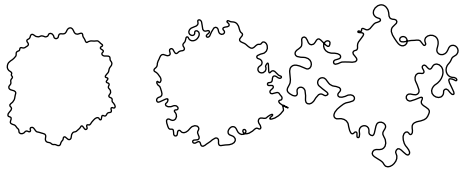
$n_0 = 8, f_n = 2, t_0 = 10, f_t = 0.6$

S = 2 S = 4 S = 8



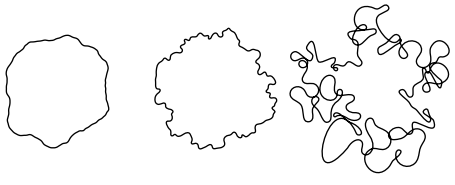
S = 3, $n_0 = 8, f_n = 2, t_0 = 10$

$t_0 = 5$ $t_0 = 10$ $t_0 = 15$



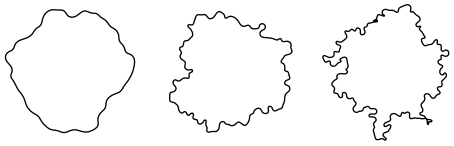
S = 3, $n_0 = 8, f_n = 2, t_0 = 10$

$f_t = 0.4$ $f_t = 0.6$ $f_t = 0.8$



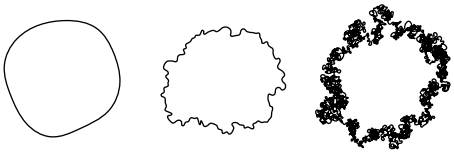
S = 3, $n_0 = 8, f_n = 2, t_0 = 10, f_t = 0.6$

$n_0 = 4$ $n_0 = 8$ $n_0 = 12$



S = 3, $n_0 = 8, f_n = 2, t_0 = 10, f_t = 0.6$

$f_n = 1.1$ $f_n = 2$ $f_n = 4$



Now we can go back to our objective. Of course we want random colors, but not completely random (it would not be pretty in general). It is useful to use palettes, so you can change easily from one set of colors to another set. Let's build a palette :

```
color MyPalette[] ; Ncolors := 5 ;
MyPalette[1] := (215/255,233/255,244/255);
MyPalette[2] := (234/255,187/255,076/255);
MyPalette[3] := (238/255,148/255,056/255);
MyPalette[4] := (199/255,066/255,033/255);
MyPalette[5] := (033/255,061/255,085/255);
```

To choose a random color, we need to choose an integer between 1 and `Ncolors`; a simple, useful and more general macro is made to choose a random integer in the interval $[mini, maxi]$:

```
vardef ranint (expr mini , maxi ) =
  floor(uniformdeviate (maxi - mini +1 ) + mini)
enddef ;
```

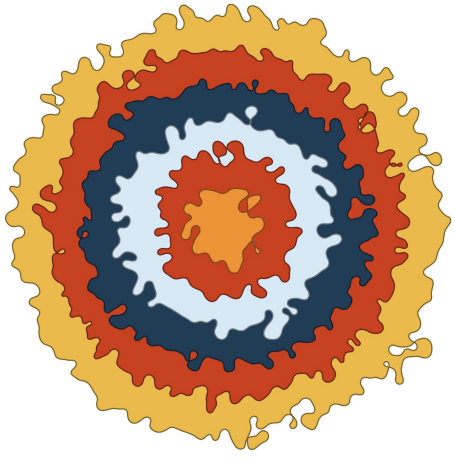
For illustration, let's fill squares of random colors in our palette:

```
for i=1 upto 2:
 for j=1 upto 8:
  fill fullsquare randomized 0.1 scaled 15
    shifted (j*20,i*20)
    withcolor MyPalette[ranint(1,Ncolors)] ;
 endfor;
endfor;
```

We have now all the elements for our project. We just need to draw *agitated* circles decreasing in size, and fill

them with different color. There is just one detail we need to pay attention to : the initial number of points used to agitate our circles should depend on the length of each path; so we add to the code in the loop `nzero := floor(arclength(P)/4.5);` the factor 4.5 is found by trial and error to obtain what we are looking for, and the function `floor` is too assure that `nzero` is an integer. In such code, we usually try to parametrize as much as possible, so it will be easier later to search optimal parameters.

```
randomseed := 1241 ; color AColor ;
path P, Q ;
NbCircles := 6 ; S := 8 ; f_n := 1.1 ;
tzero := 6 ; f_t := 0.80 ;
for c=NbCircles downto 1 :
 P := fullcircle scaled (c*30) ;
 AColor := MyPalette[ranint(1,Ncolors)] ;
 nzero := floor(arclength(P)*0.30);
 Q := agitate(P , S , nzero , f_n , tzero, f_t);
 eofill Q withcolor AColor ;
 draw Q withcolor .5[black,AColor];
endfor;
```



Now that we have succeeded, we can try to explore the possibilities of the algorithm. We can randomly generate some drawings selecting some parameters in a certain range. One could draw several pieces on the same page, but an easier way is to generate several pieces one piece per page. The complete code is below, and explanations follows.

```
\starttext
% Inclusions ------------------------------ ;
\startMPinclusions
 vardef agitate(expr apath, S, n, fn, t, ft) =
  save R , nbpoints , noiselevel ;
  path R ; nbpoints := n ; noiselevel := t ;
  R := apath ;
   for s=0 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 ;

color MyPalette[] ; Ncolors := 5 ;
MyPalette[1] := (215/255,233/255,244/255);
MyPalette[2] := (234/255,187/255,076/255);
MyPalette[3] := (238/255,148/255,056/255);
MyPalette[4] := (199/255,066/255,033/255);
MyPalette[5] := (033/255,061/255,085/255);

vardef ranint (expr mini , maxi ) =
  floor(uniformdeviate (maxi - mini +1) + mini)
enddef ;
vardef ranuni (expr mini , maxi ) =
  uniformdeviate (maxi - mini) + mini
enddef ;
\stopMPinclusions

\dorecurse{16}{ %
% MP page -- ------------------------------ ;
 \startMPpage

 randomseed := 100*#1 ;

 path O, P , Q ;
 O := fullcircle scaled 200 ;
 color AColor ;
 NbCircles := ranint(3,15) ;
 S := ranint(2,3) ; fn := ranuni(1.1,1.5) ;
 tzero := ranuni(3,8) ; ft := ranuni(0.5,0.9) ;
 CurrentColor := ranint(1,Ncolors) ;
 for c=NbCircles downto 1 :
    P := O scaled (c/NbCircles) ;
    AColor := MyPalette[CurrentColor] ;
    nzero := floor(arclength(P)*ranuni(0.2,0.8));
    Q := agitate(P , S , nzero , fn , tzero, ft);
    eofill Q withcolor AColor ;
    draw Q withcolor .5[black,AColor];

    forever:
       AnotherColor := ranint(1,Ncolors) ;
       exitif CurrentColor <> AnotherColor ;
    endfor;
    CurrentColor := AnotherColor ;
endfor;

\stopMPpage
}
\stoptext
```

Some remarks about this code:

a. Lines 2-32: functions already defined are placed between:

```
\startMPinclusions
Definitions of functions here..
\stopMPinclusions
```

b. Line 37: this instruction is a way to keep track of what random seed is giving which piece. For exemple, if you do 100 drawings and you want to reproduce only the 90th page, the same code with `randomseed := 90*100;` will suffice. This is useful to debug sometimes too.

c. Lines 35-61: the code which produce each page is between the two braces inside the `\dorecurse{16}{}`; here 16 pages will be created:

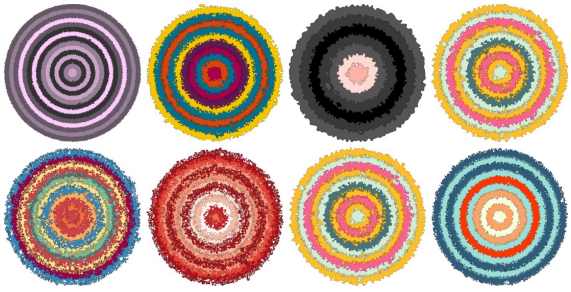
```
\dorecurse{16}{
   \startMPpage
     The code for the drawing itself
   \stopMPpage }
```

d. Lines 40,47: we have modified the previous code in order to have an algorithm able to manage a randomized path. At line 40, an original path `O` is created, here a circle, and this path is scaled down at line 47 at each step.

e. Lines 55-59: we have improved our previous algorithm by changing the color for each new circle, so two consecutive circles have not the same color; this is done in this loop by selecting a number in `(1,Ncolors)` until this result is different than the number of the current color.

Here are the 16 pages that the previous code produces:

Continuing to play with our project, it is now very easy to change the color palette. Imagine you have access to say 100 color palettes, we can choose randomly a palette (after the `randomseed` instruction on line 37), and here is a sample of what is possible (code not shown):



Now, continuing our experimentation, if we change

```
O := fullcircle scaled 200 ;                    40
```

by

```
O := fullsquare scaled 200 ;                    40
```

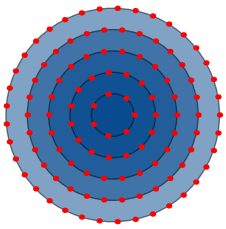
we could obtain something like this:

As you can see, once a drawing algorithm is made, it is quite easy to modify the parameters, the shapes, the colors... to explore the possibilities of the algorithm, and maybe discover an amazing creation resulting from the combination of a human idea and chance.

Project two : a sun

We can try to exploit our new function `agitate()` to create more lively pieces. We would like the piece to have the spirit of a cell, or a sun, something like this. So the strategy here is be to fill several agitated circles one above the other, like before, but this time, the border of the circles will be more chaotic, we will use more circles, and we will fill them with a transparent color.

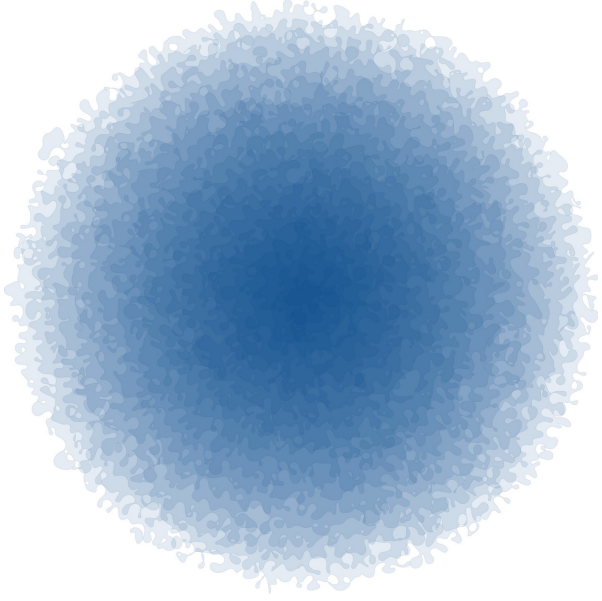
Here a sketch of the structure :



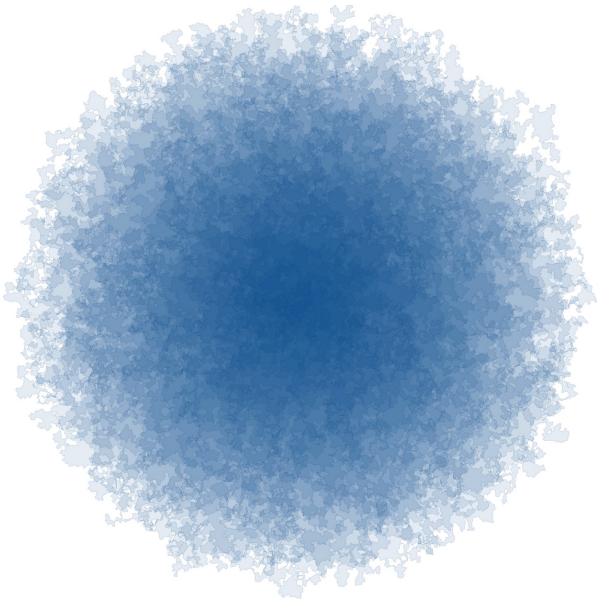
It is a matter of seconds to run this code:

```
vardef agitate(expr thepath, S, n, fn, t, ft) =
       (...)
enddef ;
path P , Q ;
color AColor ;
NbCircles := 20; S := 1; nzero:= 10; fn := 1.3;
tzero := 5; ft := 0.8;

% with \usecolors[crayola] ;
AColor := \MPcolor{MidnightBlue};

for c=NbCircles downto 1 :

 P := fullcircle scaled (c*10.5) ;
 nzero := floor(arclength(P)*0.5);
 Q := agitate( P , S , nzero , fn , tzero, ft);
 eofill Q
   withcolor transparent(1,2/NbCircles,AColor);
 draw Q  withpen pencircle scaled 0.1
   transparent(1,4/NbCircles,.90[black,AColor]);

endfor;
```

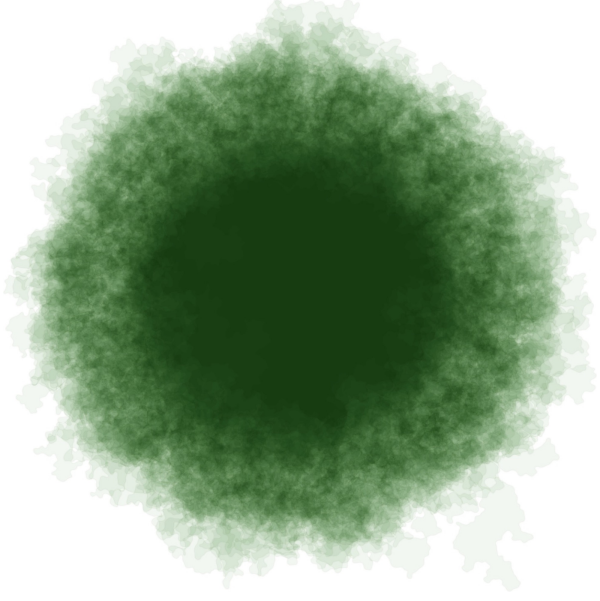
and to obtain this result :



But the borders are too smooth. So increasing the number of steps of the `agitate()` function so 15, after approximately one hour, we have this result :



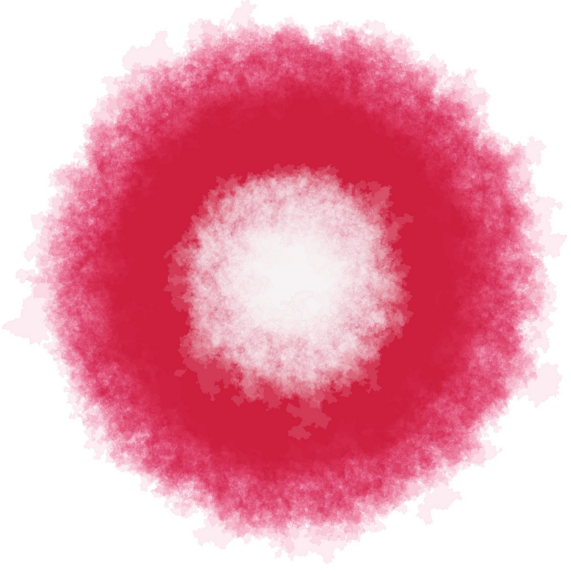
Increasing the number of circles to 40, and changing the color (to explore), but this time every circle will have the same size, we obtain this nice blurry effect :



It is tempting to try a donut by simply adding in the center a series `nbrep := 25;` of white circles. Caution : this code is very computationally intensive. if you try it, reduce the value of `S` and `nbrep`, and increase it slowly.

The code look like this :

```
path P , Q ;
color AColor ;
S := 18; nzero:= 10; fn := 1.3; tzero := 5;
ft := 0.8;

AColor := \MPcolor{Razzmatazz} ;
P := fullcircle scaled 60 ;
nzero := floor(arclength(P)*0.5);
nbrep := 25;
for rep = 1 upto nbrep:
   Q := agitate(P, S, nzero, fn, tzero, ft);
   eofill Q
     withcolor transparent(2,2/nbrep,AColor);
   draw Q  withpen pencircle scaled 0.1
     transparent(9,1/nbrep,.90[black,AColor]);
endfor;

P := fullcircle scaled 22 ;
nzero := floor(arclength(P)*0.5);
for rep = 1 upto nbrep:
   Q := agitate(P, S, nzero, fn, tzero, ft);
   eofill Q
     withcolor transparent(3,3/nbrep,white);
   draw Q  withpen pencircle scaled 0.1
     transparent(9,1/nbrep,.90[black,AColor]);
endfor;
```

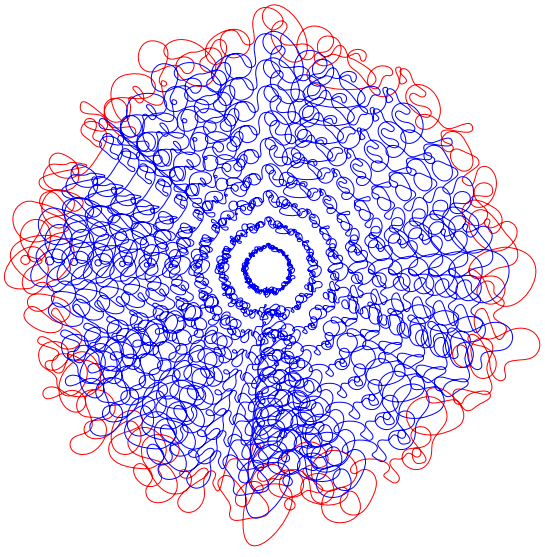


This last piece is very satisfying. One has an impression of bubbling, of life, like a gaseous planet. Satisfied, let us stop the exploration here!

Project three : a fabric

Finally we will use the same function, in a very different way, and use a simple technique to create a form that looks like a fabric. The first step is to create an *agitated* circle, a simple one this time, and scale it down a few times (here 10 times), to obtain this sketch :

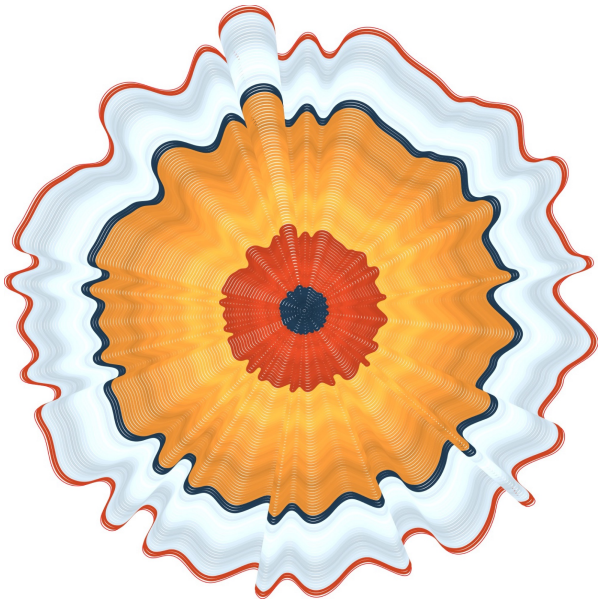
```
path P, Q;
S := 1; nzero:= 10; fn := 1.2; tzero := 15;
ft :=0.8;
P := fullcircle scaled 170 ;
nzero := floor(arclength(P)*0.5);
Q := agitate( P , S , nzero , fn , tzero, ft);
draw Q  withpen pencircle scaled 0.2
  withcolor red;
path R ; nblines := 10 ;
for i=1 upto nblines:
  R := Q scaled ((nblines-i)/nblines) ;
  draw R withpen pencircle scaled 0.2
    withcolor blue;
endfor;
```



Now we will do the same strategy, increasing the number of lines, and changing color over time, randomly of course :

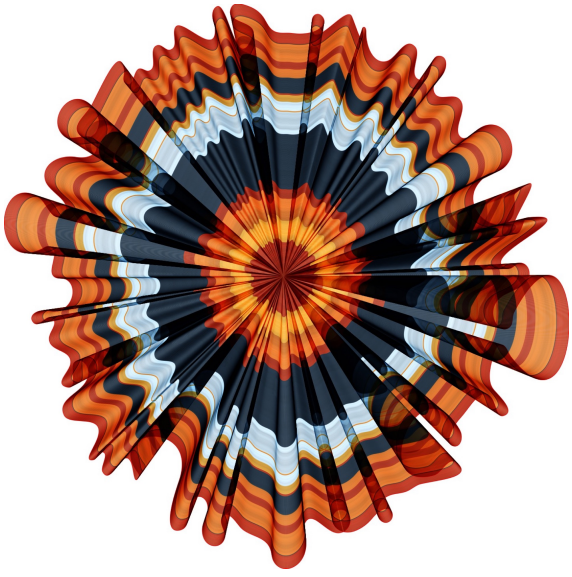
```
randomseed := 3354 ;
S :=1; nzero :=10; fn :=1.1; tzero :=15; ft :=0.8;

P := fullcircle scaled 180 ;
nzero := floor(arclength(P)*0.12);
Q := agitate( P , S , nzero , fn , tzero, ft) ;
draw Q withpen pencircle scaled 0.2 withcolor red;
path R ;
color CurrentColor , RealColor;
CurrentColor := MyPalette[ranint(1,Ncolors)];
nblines := 100 ;
```

```
for i=1 upto nblines:
  CurrentColor := CurrentColor
    randomized(0.95,1.05);
  R := Q scaled ((nblines-i)/nblines) ;
  if uniformdeviate(1) < 0.08:
    CurrentColor := MyPalette[ranint(1,Ncolors)];
  fi;
  RealColor := CurrentColor ;
  draw R withpen pencircle scaled .8
    withcolor RealColor;
endfor;
```

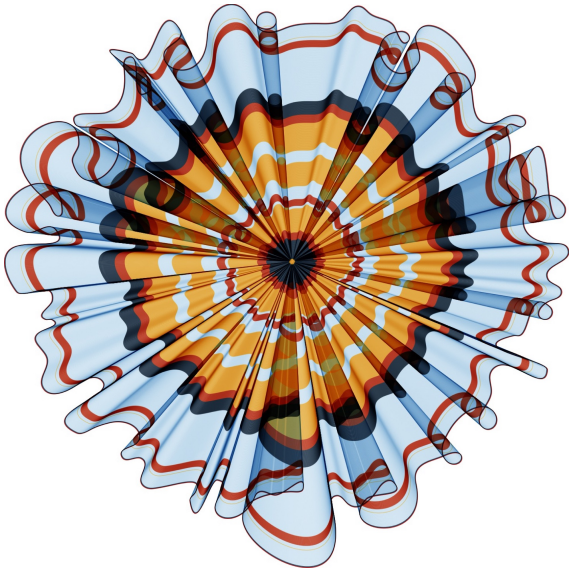


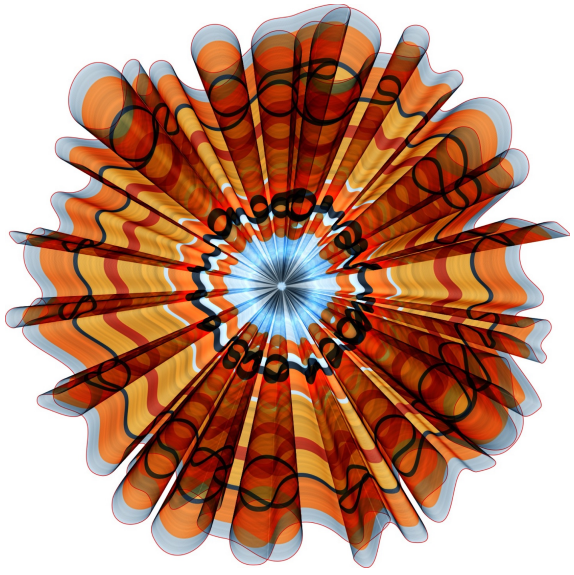
And finally, we increase dramatically the number of lines, randomized the same color en epsilon at each line, and add transparency to create relief :

```
randomseed := 2562 ;
S := 1 ; fn := 1.05 ; tzero := 20 ; ft := 0.8 ;
P := fullcircle scaled 180 ;
nzero := floor(arclength(P)*0.18);
Q := agitate( P , S , nzero , fn , tzero, ft) ;
draw Q  withpen pencircle scaled 0.2
  withcolor red;
path R ;
color CurrentColor , RealColor;
CurrentColor := MyPalette[ranint(1,Ncolors)];
nblines := 750 ;
for i=1 upto nblines:
  CurrentColor :=
    CurrentColor randomized(0.98,1.02);
  R := Q scaled ((nblines-i)/nblines) ;
  if uniformdeviate(1) < 0.025:
   CurrentColor :=
     MyPalette[ranint(1,Ncolors)];
  fi;
  RealColor := CurrentColor ;
  draw R withpen pencircle scaled .2
   withcolor transparent(2,.8,RealColor);
endfor;
```



Here are other examples changing the random seed :





And a last example decreasing the number of points `nzero`, and increasing a bit the turbulence `tzero` :

It just seems extraordinary to me that a few simple lines of code, using such elementary functions, can give such rich and varied results.

Technical addendum

Hans Hagen made a remark that the agitate code could be improved; in the `agitate()` macro we can find this instruction :

```
R := for i=1 upto nbpoints:
  point (i/nbpoints) along R
    randomized noiselevel ..
  endfor cycle ;
```

At each step of the loop `for i=1 upto nbpoints`, the instruction `along` is used, which means that the function `arclength` is called `nbpoints` times, and this function takes time. As the length of the path `R` does not change, a better way to do is to calculate this length only one time outside the loop :

```
rlength := (arclength R) / nbpoints;
R := for i=1 upto nbpoints:
  (point (arctime (i * rlength) of R) of R)
    randomized noiselevel ..
  endfor cycle ;
```

This improvement speed up the processing time significantly. A second improvement is possible, using the "double" mode improve the processing time. So all the code of this paper can be adapted like this:

```
\startMPinclusions{doublefun}
       (...)
\stopMPinclusions
\startMPpage[instance=doublefun]
       (...)
\stopMPage
```

These two modifications together speedup processing time by a factor 5 on project 2.

Conclusion

We have presented in this article how MetaFun can be used to do generative art. The `randomized` operator is simple but powerful, and can be used on several types of objects. Moreover, macros are easy to define in order to introduce new creation tools, as we did for the `agitate()` function. The ability to draw one result per page is very useful when producing a large number of results from a given algorithm, and the vector nature of the PDF output makes each drawing easily scalable and printable on a large scale. All of this makes MetaFun definitely a powerful tool, allowing to create drawings or artworks with few limitations.

Bibiography

Galenter, Phillip. 2013. *What is Generative Art? Complexity Theory as a Context for Art Theory.* In *GA2003 – 6th Generative Art Conference.*

I would like to thank Frans Goddijn for his helpful comments on the article, as well as Hans Hagen and Taco Hoekwater for their enlightenment on technical aspects of MetaFun.

Fabrice Larribe

Afscheid

Abstract
Sinds 1990 ben ik lid geweest van de NTG. Nu ik gepensioneerd ben is het gebruik van TEX zodanig verminderd dat ik mijzelf niet langer zie als een actieve gebruiker. In dit artikel kijk ik terug op ruim 30 jaar TEX en met name LaTEX gerelateerde activiteiten.

De aanloop

In de periode tot ongeveer 1990 hield ik mij incidenteel bezig met het elektronisch opmaken van documenten. Hiervoor maakte ik gebruik van de in de UNIX-wereld bekende systemen TROFF/NROFF, Eqn en Tbl. TROFF is bedoeld om documenten te kunnen produceren met behulp van foto-typesetters en NROFF is de variant die gebruikt kon worden op andere printers, zoals matrixprinters, daisywheelprinters (wie kent ze nog?) en laserprinters. Van deze programmatuur was een commerciële versie voor de PC beschikbaar.

In 1989/1990 was ik, namens het CPB, betrokken bij het opzetten en schrijven van een gebruikershandleiding voor het gebruik van de Control Data mainframe computers van het ECN. Gerard van Nes, vanaf het eerste uur betrokken bij de NTG, was de trekker van dit project en de drijvende en coördinerende kracht achter dit gezamenlijke project. Besloten werd om voor deze uitgebreide handleiding LaTEX als tekstopmaaksysteem te gebruiken. Door deze keuze was ik gedwongen om mij in de TEX-wereld te verdiepen en werd dan ook spoedig lid van de NTG en niet lang daarna bestuurslid. Mijn activiteiten binnen de NTG lagen met name op het gebied van TEX-implementaties voor PCs en werkzaamheden in het redigeren van de NTG-Maps onder leiding van Gerard van Nes.

De beschikbaarheid van TEX buiten de wereld van mainframes en mini-computers was destijds nog beperkt en het gebruik was beperkt tot de academische wereld.

TEX-distributies voor PCs

Eenmaal actief binnen TEX-gemeenschap heb ik mij met name beziggehouden met beschikbaar maken van TEX-implementaties voor PCs en dan met name op MS-Dos computers. In het begin bestond de mogelijkheid voor NTG-leden om TEX distributies voor PC op $5\frac{1}{4}$-inch floppy disks te krijgen. Destijds waren er meerdere min of meer complete distributies beschikbaar waaronder SbTeX en DosTex.

Als een poging van de NTG om het gebruik van TEX te propageren heb ik in die tijd een werkende, zij het beperkte distributie gemaakt op basis van de beschikbare systemen. Deze versie paste op twee 51/4 inch floppy disks. Deze versie was bedoeld als demonstratie-systeem en derhalve met name beperkt in het aantal beschikbare lettertypen, alleen de C-fonts. We hebben op een NTG-bijeenkomst deze versie ook gepresenteerd aan Barbara Beeton en later ook aan Donald Knuth. Zoals gezegd dit was slechts een systeem om een deel van de mogelijkheden van TEX te presenteren.

Een aanzienlijke verbetering was de door Eberhard Mattes geproduceerde emTEX-distributie. Toen dit eenmaal voldoende stabiel was werd deze versie door de NTG als standaard PC-versie verkozen. emTEX werd door de NTG rondgestuurd aan belangstellenden door een doosje met benodigde $3\frac{1}{2}$-inch diskettes.

Door Wietse Dol en Erik Frambach werd in de tweede helft van de 90-er jaren van de vorige eeuw als NTG-activiteit het op CD-roms gebaseerde 4TEX for Windows geproduceerd.

Mijn overige NTG-activiteiten

Naast mijn activiteiten om de PC-distributies te verspreiden ben ik een aantal jaren bestuurslid geweest en als zodanig betrokken geweest bij de voorbereiding en productie van de MAPS. Zoals bij uitgevers wel bekend is het produceren van een tekstdocument waaraan verschillende auteurs werken soms een tijdrovende en onder tijdsdruk staande zaak. Zeker als aan bijdragen geen stringente eisen worden gesteld aan de wijze van opmaak van de manuscripten. Met name het omzetten van teksten van *native* TEX naar de voor de MAPS gehanteerde LaTEX-layout heeft soms de nodige hoofdbrekens gekost. Naast de MAPS heeft de NTG in overleg met David Salomon ook een eerste versie van diens boek —de syllabus van de cursus die hij voor de NTG heeft gegeven— geproduceerd.

Terugblik

Op basis van het door Donald Knuth ontwikkelde systeem is in de loop van de tijd een heel arsenaal aan kwalitatief hoogstaande systemen ontwikkeld. Het gaat dan niet alleen om macro-pakketten voor LaTEX maar bijvoorbeeld ook om het door Hans Hagen ontwikkelde ConTEXt.

Jarenlang heb ikzelf in door WYSIWYG-systemen overheerste omgevingen gewerkt. Wanneer er geen interacties met andere auteurs waren dan kon ik mijn eigen gang gaan en publicaties, zoals mijn proefschrift, in LaTEX produceren. De uitwisseling van in TEX opgemaakte teksten met WYSIWYG-systemen is lastig en vergt veel kennis en handwerk, waardoor gebruik van TEX ondanks de kwaliteiten altijd beperkt zal blijven. De kwaliteiten van op TEX gebaseerde systemen blijken bij het opmaken van complexe documenten. Mijn eigen ervaring met een rapport met tientallen tabellen en vele figuren is dat dit soort complexe documenten betrekkelijk eenvoudig en efficiënt in een op TEX gebaseerd systeem is op te zetten. Dergelijke documenten van tientallen pagina's met vele tabellen en figuren kunnen in een WYSIWYG-omgeving slechts moeizaam en met veel handwerk worden opgezet.

Voor mij is een eind gekomen een een interessante en leerzame periode.

Jos Winnink