New directions in math fonts

Hans Hagen, Mikael P. Sundqvist

1 Introduction

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

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

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

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

2 Tweaks, also known as goodies

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

$$\hat{7}$$

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

The fixers is another category, and the names indicate what gets fixed. Tweaks like these take lists of code points and specific properties to fix. We could leave it to your imagination what

```
fixaccents fixanchors fixellipses
fixoldschool fixprimes fixradicals
fixslashes
```

do, but here are some details.

Starting with `fixaccents`: Inconsistencies in the dimensions of accents make them jump all over the place so we normalize them. We support horizontal stretching at the engine level.

$$\widehat{a+b+c+d} = \widehat{u+v+w+x+y}$$

This required only a few lines of code, thanks to scaling features that were already present.

`fixanchors`: Anchors can be off so we fix these to look better, especially on italic shapes. We make sure that the automated sizing works consistently, as this is driven by width and overshoot.

`fixellipses`: Several kind of ellipses can be inconsistent with each other as well as with periods (both shape- and sizewise) so we deal with that.

`fixoldschool`: TEX (TFM) fonts have a limited set of widths, heights, and depths. We need to fix for instance fences of various size because we want to apply kerns to scripts on the four possible corners, for which we need to know the real height and depth,

`fixprimes`: Discussing primes would take many paragraphs so we stick to mentioning that they are a mess. We now have native prime support in the engine, and we assume properly dimensioned symbols to be used.

`fixradicals`: TFM dimensions for the parts of a radical (e.g., square root) sign are, shall we say, unusual. Let's fix them.

`fixslashes`: Slashes are used for skewed fractions so we'd better make sure they are set up right.

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

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

doi.org/10.47397/tb/43-3/tb135hagen-mathchange

just like we look up a smaller variant when we're in compact font mode (a ConTEXt feature).

$$\sum_{\text{س}=\text{ب}}^{\text{ج}} \text{ى}^{\text{س}} = \text{ى}^{\text{ب}} \frac{\text{١} - \text{ى}^{\text{ج}-\text{ب}+\text{١}}}{\text{١} - \text{ى}} \qquad (\text{ى} \neq \text{١})$$

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

$$a + \alpha + \alpha$$

For this we ship a companion font. On our disks (and in the distribution) you can find, in the directory `/tex/texmf-fonts/fonts/data/cms/companion`:

```
RalphSmithsFormalScript-Companion.otf
TeXGyreBonumMath-Companion.otf
XITSMath-Companion.otf
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Font parameters

At some point in the upgrading of the math machinery we discussed some of the inconsistencies between the math constants of the XITS and STIX fonts. Now, one has to keep in mind that XITS was based on a first release of STIX that only had Type 1 fonts so what follows should not to be seen as criticism, but more as observations and reason for discussion, as well as a basis for decisions to be made.

One thing we have to mention in advance: we often wonder why weird and/or confusing stuff in math fonts goes unnoticed. We have some ideas:

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

We do not claim to be perfect, but we care for details, so let's go on. Table 1 shows the math constants as they can be found in the STIX (two) and XITS (one) fonts. When you typeset with these fonts you will notice that XITS is somewhat smaller, so two additional columns show the values used to compensate for the axis height and accent base height. For the relevance column: (1) 'mandatory' means a design-related value the font designer must supply; (2) 'optional' means apparently redundant values that would normally be identical; (3) a blank cell (the vast majority) means a value likely needed to be configured at the macro/document level.

As you can see in the table, very few values are the same. So, what exactly do these constants tell us? You might even wonder why they are there at all. Just think of this: we want to typeset math, and we have an engine that we can control. We know how we want it to look. So, what do these constants actually contribute? Plenty relate to the height and depth of the nucleus and/or the axis. The fact that we have to fix some in the goodie files, and the further fact that we need more variables that control positioning, makes for a good argument to just ignore most of the ones provided by the font, especially when they seem somewhat arbitrary. Can it be that font designers are just gambling a bit, looking at another font, and starting from there?

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

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

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

Internally we use different names for variables, simply because the engine has some parameters that

Table 1: OpenType math parameters, compared; bold indicates an unchanged value. See text for explanation of relevance.

constant	STIX	XITS	base	axis	relevance
`AccentBaseHeight`	450	480	**480**	464	optional**
`AxisHeight`	250	258	267	**258**	mandatory
`DelimitedSubFormulaMinHeight`	1500	1325	1600	1548	
`DisplayOperatorMinHeight`	1450	1800	1547	1496	
`FlattenedAccentBaseHeight`	662	656	706	683	optional**
`FractionDenominatorDisplayStyleGapMin`	198	150	211	204	
`FractionDenominatorDisplayStyleShiftDown`	700	640	747	722	
`FractionDenominatorGapMin`	66	68	70	**68**	
`FractionDenominatorShiftDown`	480	585	512	495	
`FractionNumeratorDisplayStyleGapMin`	198	150	211	204	
`FractionNumeratorDisplayStyleShiftUp`	580	640	619	599	
`FractionNumeratorGapMin`	66	68	70	**68**	
`FractionNumeratorShiftUp`	480	585	512	495	
`FractionRuleThickness`	66	68	70	68	optional
`LowerLimitBaselineDropMin`	600	670	640	619	
`LowerLimitGapMin`	150	135	160	155	
`MathLeading`	150	150	160	155	
`MinConnectorOverlap`	50	100	53	52	mandatory
`OverbarExtraAscender`	66	68	70	**68**	
`OverbarRuleThickness`	66	68	70	**68**	optional*
`OverbarVerticalGap`	198	175	211	204	
`RadicalDegreeBottomRaisePercent`	70	55	75	72	mandatory
`RadicalDisplayStyleVerticalGap`	186	170	198	192	
`RadicalExtraAscender`	66	78	70	68	
`RadicalKernAfterDegree`	−555	−335	-592	-573	
`RadicalKernBeforeDegree`	277	65	295	286	
`RadicalRuleThickness`	66	68	70	**68**	
`RadicalVerticalGap`	82	85	87	**85**	
`ScriptPercentScaleDown`	75	70	80	77	
`ScriptScriptPercentScaleDown`	60	55	64	62	
`SkewedFractionHorizontalGap`	300	350	320	310	
`SkewedFractionVerticalGap`	66	68	70	**68**	
`SpaceAfterScript`	41	40	44	42	
`StackBottomDisplayStyleShiftDown`	900	690	960	929	
`StackBottomShiftDown`	800	385	853	826	
`StackDisplayStyleGapMin`	462	300	493	477	
`StackGapMin`	198	150	211	204	
`StackTopDisplayStyleShiftUp`	580	780	619	599	
`StackTopShiftUp`	480	470	512	495	
`StretchStackBottomShiftDown`	600	590	640	619	
`StretchStackGapAboveMin`	150	68	160	155	
`StretchStackGapBelowMin`	150	68	160	155	
`StretchStackTopShiftUp`	300	800	320	310	
`SubSuperscriptGapMin`	264	150	282	272	
`SubscriptBaselineDropMin`	50	160	53	52	
`SubscriptShiftDown`	250	210	267	258	
`SubscriptTopMax`	400	368	427	413	
`SuperscriptBaselineDropMax`	375	230	400	387	
`SuperscriptBottomMaxWithSubscript`	400	380	427	413	
`SuperscriptBottomMin`	125	120	133	129	
`SuperscriptShiftUp`	400	360	427	413	
`SuperscriptShiftUpCramped`	275	252	293	284	
`UnderbarExtraDescender`	66	68	70	**68**	
`UnderbarRuleThickness`	66	68	70	**68**	optional*
`UnderbarVerticalGap`	198	175	211	204	
`UpperLimitBaselineRiseMin`	300	300	320	310	
`UpperLimitGapMin`	150	135	160	155	

OpenType math does not. So for `bigopspacing5`, we have `limit_above_kern` and `limit_below_kern`.

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

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

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

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

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

Is there a reason why `UnderbarRuleThickness` and `OverbarRuleThickness` should differ? If not, then we only need a variable that somehow tells us what thickness fits best with the other top and bottom accents. It is quite likely the same as the `RadicalRuleThickness`, which is needed to extend the radical symbol. So, here three constants can be replaced by one design-related one. The parameter `FractionRuleThickness` can also be derived from that, but more likely is that it is a quantity that the macro package sets up anyway, maybe related to rules used elsewhere.

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

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

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

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

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

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

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

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

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

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

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

How about StackBottomShiftDown, StackBottomDisplayStyleShiftDown, StackDisplayStyleGapMin, StackGapMin, StackTopDisplayStyleShiftUp, StackTopShiftUp? And aren't these more for the renderer to decide: StretchStackBottomShiftDown, StretchStackGapAboveMin, StretchStackGapBelowMin, StretchStackTopShiftUp?

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

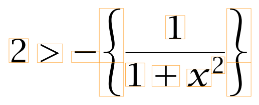
Just for the record, here are the (font-related) ones we added so far. A set of prime-related constants similar to the script ones: PrimeBaselineDropMax, PrimeRaisePercent, PrimeRaiseComposedPercent, PrimeShiftUp, PrimeShiftUpCramped, PrimeSpaceAfter, PrimeWidthPercent.

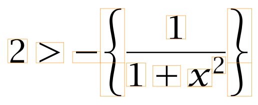
We also added SpaceBeforeScript just because we want to be symmetrical in the engine where we also have to deal with prescripts.

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

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

Setting these parameters right is not trivial, and also a bit subjective. We might, for instance, assume that the math axis is set right, but alas, when we were fixing the less and greater symbols in Lucida Bright Math, we found that all symbols were designed for a math axis of 325, instead of the given value 313, and that difference can be seen. If you look closely, the points on the greater than sign and the braces are slightly below the minus sign in "Old Lucida" on the left, and aligned completely on the right. (The greater than sign is also larger in size. See the accompanying article on Lucida for more examples and discussion of this particular font.)


Old Lucida


New Lucida

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

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

4 Metrics, especially italic corrections

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

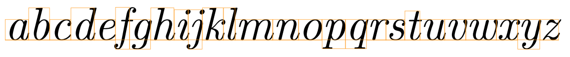
Given the times TeX evolved in, the format is rather compact. For instance, the height, depth and italic correction are shared and indices to three shared values are used. There can be only 16 distinct heights, 16 depths and 64 italic corrections. That way much fits into a memory word.

The documentation tells us that "The italic correction of a character has two different uses. (a) In ordinary text, the italic correction is added to the width only if the TeX user specifies '\/' after the character. (b) In math formulas, the italic correction is always added to the width, except with respect to the positioning of subscripts."

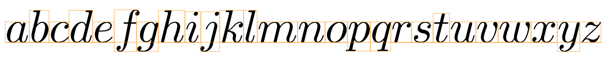
It is this last phenomenon that gives us some trouble with fonts in OpenType math. The fact that traditional fonts cheat with the width and that we add and selectively remove or ignore the correction makes for fuzzy code in LuaTeX, although splitting the code paths and providing options to control all this helps a bit. In LuaMetaTeX we have more control but also expect an OpenType font. In OpenType math there are italic corrections, and we even have the peculiar usage of it in positioning limits. However, the idea was that staircase kerns do the detailed relative positioning.

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

Latin Modern Roman (text) italic:



Latin Modern Roman math italic:



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

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

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

It is for that reason that an engine can produce reasonable results for fonts that either provide italics or provide kerns for anchoring: having both on the same glyph would mean troubles. It means that we can configure the engine options to add italic correction as well as kerns, assuming distinct usage of those features. For a font that uses both we need to make a choice (this is possible, since we can configure options per font). But that will certainly not lead to math that is always nicely typeset. In fact, without tweaks many fonts will still look right because in practice they use some mixture. But we are not aiming at partial success, we want all to look good.

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

4.1 Spacing

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

But glyph properties cannot be easily used here because we are still dealing with characters when the engine transforms the noad list into a node list. So, when we discussed this, we started wondering how the engine could know about a specific shape (and tilt) property at all, and that brought us to pondering about an additional axis of options. We already group characters in classes, but we can also group them with properties like `tilted`, `dotless`, `bold`. When we pair atoms we can apply options, spacing and such based on the specific class pair, and we can do something similar with category pairs.

It boils down to, for instance, a new `\mccode` that binds a character to a category. Then we add a command like `\setmathcategorization` (analogue to `\setmathspacing`) that binds options to pairs of categories. An easier variant of this might be to let the `\mccode` carry a (bit)set of options that then get added to the already existing options that can be bound to character noads as we create them. This saves us some configuration. Deciding what suits best depends on what we want to do: the fact that TEX doesn't do this means that probably no one ever gave it much thought, but once we do have this mechanism it might actually trigger demand, if only by staring at existing documents where characters of a different kind sit next to each other (take this 'a' invisible times 'x'). It would not be the first time that (in ConTEXt) the availability of some feature triggers creative (ab)usage.

4.2 Moving towards kerns

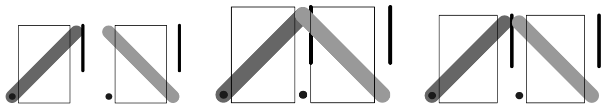
Because the landscape has settled, because we haven't seen much fundamental evolution in OpenType math, because in general TEX math doesn't particularly evolve, and because ConTEXt in the past has not been seen as suitable for math, we can, as mentioned before, basically decide what approach we follow.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

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

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

5 Kerning

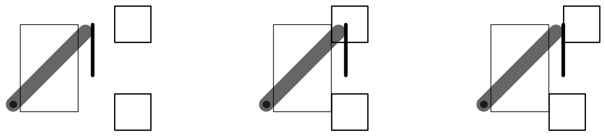
In the next pictures we will try to explain more visually what we have in mind and are experimenting with as we write this. In the traditional approach we have shapes that can communicate the width, height, depth and italic correction to the engine so that is what the engine can work with. The engine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


two characters width only with italic

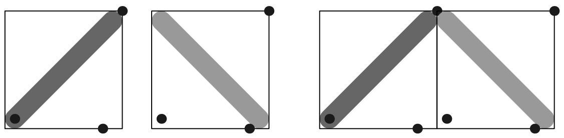
In this graphic we show two pseudo-characters. Each shown bounding box indicates the width as seen by the engine. An example of such a shape is the math italic 'f', and as it is used a lot in formulas it is also one of the hardest to handle when it comes to spacing: in nearly all fonts the right top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

When we put two such characters together they will overlap, and this is why we need to add the italic correction. That is also why the TEX documentation speaks in terms of "always add the italic correction to the width". This also means that we need to remove it occasionally, something that you will notice when you study for instance the LuaTEX source, that has a mix of traditional and OpenType code paths. Actually, compensating can be done either by changing the width property of a glyph node or by explicitly adding a kern. In LuaMetaTEX we always add real kerns because we can then trace better.

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

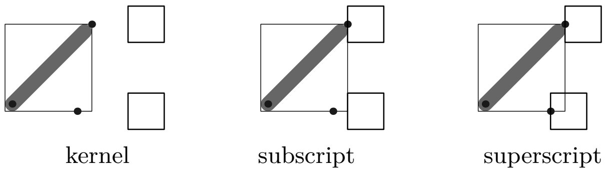

kernel subscript superscript

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


two characters width only

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

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

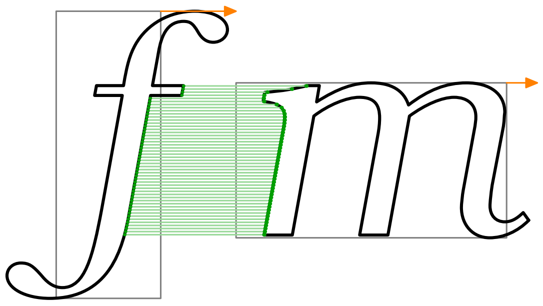


Finally we add the scripts here. This time we position the superscript and subscript at the top and bottom anchors. The bottom anchor is, as mentioned, the old italic correction, and the top one currently just the edge. And this is what our next project is about: identify the ideal anchors and use these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

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

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

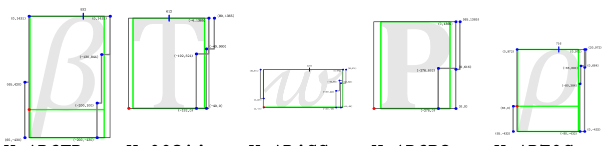
The approach that we follow is somewhat interactive. When working on the math update we like to chat (with Zoom) about these matters. We discuss and explore plenty and with these kerns we do the same. Because MetaPost produces such nice and crisp graphics, and because Metafun is well-integrated into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



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

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

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


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

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

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

6 Conclusion

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

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

This is how we plan to go forward:

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

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

◇ Hans Hagen
Pragma ADE

◇ Mikael P. Sundqvist
Department of Mathematics
Lund University
Box 118
221 00 Lund
Sweden
`mickep (at) gmail dot com`