BEYOND

luametatex & context lmtx

Table of contents

1 Introduction

This is the eights wrapup of the LuaTeX and LuaMetaTeX development cycle. The last one was 'ontarget' and focused on what we did when the engine got mature. This time we zoom in on developments that go a bit beyond what we originally planned. One can argue that for instance some of the math extensions should have ended up here but for us a turning point was when additional par passes became stable, which was around the time of the 2024 ConTeXt meeting. We'll see what comes after that.

Most of the chapters in this document were first published in TugBoat, in which case we have a (year or so) delay in including it here; just become a tug member if you want it sooner. We therefore want to explicitly mention that Karl Berry did an amazing job on copy-editing and getting it production ready in a way that we can still feed back fixes to the text. He and Barbara Beeton not only improve the English but also catch glitches in our explanations. It just gets better, so, thanks to the editorial team!

Hans Hagen[1]
Hasselt NL
August 2024^{++}

---

[1] Various chapters in this document are co-authored by Mikael Sundqvist and/or Keith McKay.

2 A new take on paragraphs

2.1 Introduction

The excellence of the Knuth–Plass algorithm for breaking paragraphs into lines is one of the reasons for the success of T_EX. It is very fast (built upon dynamic programming) and powerful (it can combine both justification and hyphenation in one go). The algorithm is built to use so-called demerits in a cost function to determine the optimal breakpoints.

The paragraph builder is however limited to at most three runs over each paragraph to get the job done. In this article we will describe some new ideas and tools regarding the process of paragraph building. What we describe is already available in LuaMeta-T_EX and ConT_EXt. The main new feature is that it is now possible to have an arbitrary number of runs over each paragraph and to configure them independently.

If T_EX is an example of "The Art of Programming", then we might approach some of its building blocks as pictures. These come in flavors; some are concrete and show a scene that leaves no doubt about what is pictured. Others are more abstract and can let us imagine or experience something and anyway leave interpretation to the viewer. When old paintings are restored quite often layers under the top layer show something different. The canvas might have been repurposed or we can see intermediate (even different) versions of what the final result is. Modern paintings can use paint that was hip at that moment but was not durable over a long term, so drastic measures are needed.

The par builder code in LuaMetaT_EX has all these aspects: features were added, some on top of others, the code and algorithm is open for interpretation, some tricks relate to the toolkit used. This makes fundamental extensions hard and a rewrite has the danger of losing compatibility. To quote from Knuth's T_EX source:

> "This particular part of T_EX was a source of several subtle bugs before the correct program logic was finally discovered; readers who seek to improve T_EX should therefore think thrice before daring to make any changes here."

The original T_EX par builder is (at least for us) not something that immediately reveals its workings. Since the logic is a bit fuzzy to us, we have to be able to analyze what we see. There are many parameters like `\pretolerance` and `\tolerance`, as well as badness, penalties, demerits and all of these plus slack in a line leads to a conclusion about how bad breakpoints are. Add to that comparing neighboring lines with respect to how much the applied spacing differs.

On top of that ε-TeX added some layers (like last line related) and pdfTeX added even more due to expansion and protrusion. We can see some remnants of Omega (Aleph) like local boxes, too. The LuaTeX approach separated the hyphenation, ligature building and kerning from the main task. Then LuaMetaTeX added more control, various new features, and node list normalization from the perspective of access by Lua.

So, the whole picture becomes more complex and abstract over time. And one indeed has to be careful when adding new features to it. Some comments in the source indicate that coming to the right solution has been a step-wise process. Just like painters made their own paint we have all kinds of helpers. We can trace what TeX does, and what solution was considered best. We can do that visually as well as via extensive logging. These helpers have been invaluable in the work to extend the paragraph builder.

The idea to use more runs over the paragraph is however not new. In D.E. Knuth's *Digital Typography* we can read the following:

> "On the other hand, some paragraphs are inherently difficult, and there is no way to break them into feasible lines. In such cases the algorithm we have described will find that its active list dwindles until eventually there is no activity left; what should be done in such a case? It would be possible to start over with a more tolerant attitude toward infeasibility (a higher threshold value for the adjustment ratios). TeX takes the attitude that the user wants to make some manual adjustment when there is no way to meet the specified criteria, so the active list is forcibly prevented from becoming empty by simply declaring a breakpoint to be feasible if it would otherwise leave the active list empty. This results in an overset line and an error message that encourages the user to take corrective action."

Maybe it was the limitations of computers at that time that prevented more runs? So, given the faster computers and already opened-up code base, which permits extensive visual tracing, we decided to play with multiple passes, a mechanism that will be discussed below. When documenting this we occasionally went back to Knuth's descriptions, like the ones above, and admit that some started making sense only in retrospect. For instance the "so the active list is forcibly prevented from becoming empty by simply declaring a breakpoint to be feasible" action was something that we had to circumvent in order to let additional passes kick in at all. We still learn.

2.2 The traditional par builder

Before we move on and discuss the possibility of using more paragraph passes, we will discuss in a bit more detail how the traditional par builder works. This will help us to better understand the various extensions in LuaMetaTeX.

For good order a paragraph is a horizontal list, wrapped over lines. This happens either in a `\vbox` or in the main vertical list (page). When such a list is broken, TeX has to keep track of the current width of a line. That width can change depending on the so-called par shape or hanging indentation.

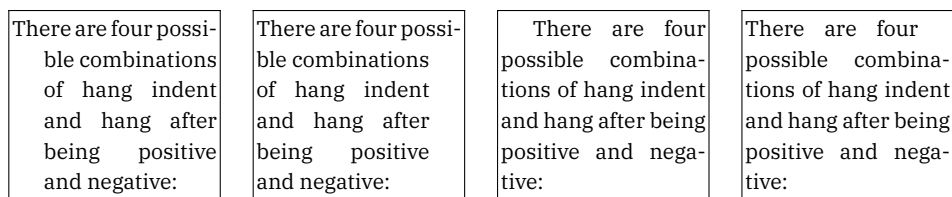| There are four possi-ble combinations of hang indent and hang after being positive and negative:	There are four possi-ble combinations of hang indent and hang after being positive and negative:	There are four possible combina-tions of hang indent and hang after being positive and nega-tive:	There are four possible combina-tions of hang indent and hang after being positive and nega-tive:

Figure 2.1　hangindent

We show the hanging indentation in Figure 2.1, which makes clear that it adds a constraint. We can also have an indentation on the first line, left and right par fill skip (last line) as well as left and right init skip (first line). Then there are left and right skip but these are the same for every line. All this means that the par builder has to keep track of the current line, in order to set the current width.

If we render table cells, or captions, or a narrow quote, or text flowing around an image, the width can be a limiting factor and combined with penalizing hyphenation, multiple hyphens in a row, specific demands like inline math, the solution space can become cramped but we will notice that the engine can quite well deal with these situations, unless of course we leave no room, for instance by setting every penalty that plays a role to 10000 or demerits to the maximum number possible.

It is also good to keep in mind that a macro package can have features that interfere with what otherwise would be a pristine paragraph. Think of a forced linebreak (`\crlf`), binding words (using `~`), switching fonts and thereby spacing, ligatures and kerning, changing to a language with fewer or more short words, compound words and possibly different hyphenation rules, verbatim, which normally runs wider and doesn't hyphenate.

TeX first tries to break the paragraph list into lines without using hyphenation, within the constraints of the `\pretolerance` value. If this fails a second pass will use `\tolerance` as the constraint, with hyphenation enabled. The verdict is also influenced by various penalties, for instance those that penalize one or more hyphens at the end of lines. If the outcome is still not right, a third pass permits `\emergencystretch` to be applied.

The decision to enter a next pass is determined by a valid result. So, if a pass processes the whole list within the constraints we have a result and no further passes are done. When there is no result, the next pass will be entered. We can skip the first pass by setting `\pretolerance` to −1, and the third pass won't happen if we have no

emergency stretch. It is important to have a final pass, because TeX has to make sure to provide a result. Thus, we can have the following cases.

1. pretolerance tolerance
2. pretolerance tolerance stretch
3. tolerance
4. tolerance stretch

Take the first situation: we run the pretolerance pass, if there is a valid result we quit, otherwise we run the tolerance pass which is tagged final and therefore will be forced to always have a result by dealing with troublesome breakpoints. There is no third pass because emergency stretch is zero. This is where the majority of TeX users end up.

In the second case we can succeed after the pretolerance pass and quit, or carry on with the tolerance pass, where again we can succeed or carry on. The stretch pass is the final one and it must result in something.

The third case is final right from the start so the first pass will always result in something, no matter how bad. The fourth case can succeed after the tolerance pass but can carry on with the last pass, which then must return a result.

Traditionally, TeX can break lines

- at glue (after words, not usually inside math),
- at a kern followed by glue,
- at a discretionary (hyphenation),
- due to a penalty (also inside math).

We will look at many examples below, and for them we will (re)use a paragraph written by the famous mathematician and physicist P.A.M. Dirac.[2]

As TeX runs over a paragraph, a badness value is attached to each possible breakpoint. With the default parameter settings of `\pretolerance` to `100` and `\tolerance` to `200`, many of the possible breakpoints are discarded, since the badnesses attached to them are greater than the tolerance; they would simply lead to non-optimal (within our measurement of tolerance) lines. In ConTeXt we can use the pair of commands `\startshowbreakpoints` and `\stopshowbreakpoints` to show the possible breakpoints; see Figure 2.2.

---

[2] From his article "Pretty Mathematics", *Internat. J. Theoret. Phys.* vol. 21, no. 8–9, pp. 603–605, 1981/82. Presented at the Dirac Symposium, Loyola University, New Orleans, May 1981.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2 ×|2| matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1,|σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so|as to|form|$\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This|was|an| exciting|result, but what use could one make of it?
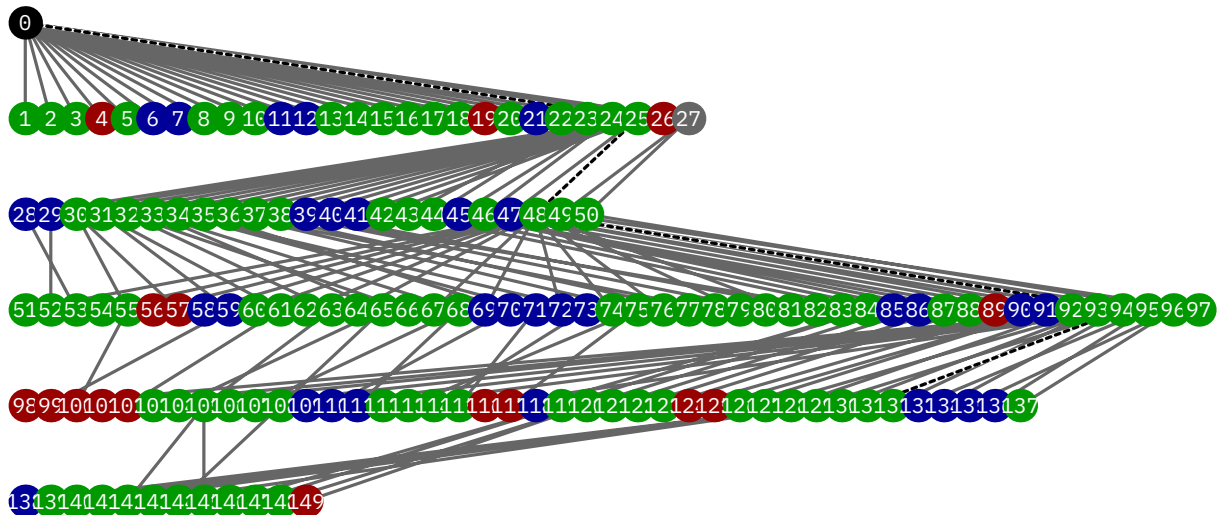
Figure 2.2   Feasible breakpoints are marked with vertical bars. Here we are using the default settings \ (and \).

In fact, each breakpoint (except the first one, which sits at the start of the paragraph and has index 0) points to a previous breakpoint; a tree is built. With \showbreak-points we can get some information about the breakpoints that TeX kept until it was time to make the choice; see Figure ??. We will discuss the details a bit later when we have introduced the relevant concepts.

```
1   1   0  13     529   loose   glue        11   5   0   31710  tight   glue     14   7 4 1
    2   0   3  490169   decent  penalty   4 12   7  54    5238  loose   glue     15   8 4 1
    3   0  45    3025   tight   math        13   8  14    1515  loose   glue     16   7 4 1
2   4   1   7     818   decent  glue        14   7  14    1242  decent  glue     17  10 6 3
    5   3   6   25781   decent  glue        15   8   8    1039  decent  glue     pass      : 1   demerits  : 1139
    6   3  38   27829   tight   glue        16   7   8    2511  tight   glue     subpass   : P   looseness :    0
3   7   4   8    1142   decent  glue        17  10   1   28050  decent  glue     subpasses : 0
    8   4   1     939   decent  glue     11  5 3
    9   4  15   12054   tight   glue     12  7 4 1
   10   6   0   27929   decent  glue     13  8 4 1
```

Table 2.1

We can also draw a representation of the tree with \drawbreakpoints, where each line represent the possible breakpoints for different lines; see Figure 2.3. Some of the nodes are kept by TeX until the end, even though no later node points back to them.

Figure 2.3   The tree TeX has after going over the paragraph, with the selected path dashed.

By setting the parameters \pretolerance to -1 and \tolerance to 10000, we can fool TeX into viewing all possible breakpoints in a paragraph as feasible, not throwing away any of them; see Figures 2.4 and 2.5.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.4   All possible breakpoints considered by TeX.

Figure 2.5   All possible breakpoints, drawn in a cramped manner. We observe that among all solutions, some are one line longer than the selected solution.

We have mentioned badness and tolerance, and that breakpoints are discarded if their badness is larger than the tolerance. We will next explain how badness values are calculated. To do that, we first need the *adjustment ratio*.

When TeX starts to run over a paragraph, it knows the desired length of each line. Usually these lengths are fixed, but they can vary a bit depending on (hanging) indentation, or even more advanced par shapes. While running over the paragraph, there will be a few active nodes. At the start, it is only the one that we have marked with a 0 in the upper left of the figures. Then every possible breakpoint points back to node 0. When TeX goes on, it will check for each possible breakpoint in turn, whether it, when pointing back to 0, will give a line that is okay according to the rules set up. If so, it will add that point to the list of active nodes and move on. Once we come to a point where the first line would be more than completely filled, it will deactivate node 0. Hopefully, there will be new active nodes to test new possible breakpoints against (if not, the run fails). Other active nodes will also be deactivated in a similar way as TeX moves on.

When TeX creates an active node it also creates a so-called passive node, which carries additional information. It is the passive nodes that point to the previous breakpoints and build the tree. There can be multiple nodes pointing to a node, but only the one with the fewest demerits in each fitness class is kept. If an active node is deactivated, the passive nodes are not cleaned up. (This is also why we can generate the tree graphics.)

11   A new take on paragraphs

We now assume that we have a new possible breakpoint, and an active one that it might be able to point back to. Let ℓ be the desired length of the corresponding line (which is known). Let L be the total *natural width* of what we have so far (without stretch and shrink), calculated from the active breakpoint, considered at the moment, up to the current candidate. Also, let $Y > 0$ be the total *stretchability* and $Z > 0$ be the total *shrinkability*. (Negative values are possible but we leave them out of this discussion.) Define the adjustment ratio r as

$$
r = \begin{cases}
(\ell - L)/Y, & L < \ell; \\
0, & L = \ell; \\
(\ell - L)/Z, & L > \ell.
\end{cases}
$$

The closer r is to 0, the less stretch or shrink is needed. When $r = -1$, all available shrink is asked for, and when $r = 1$ all stretch is needed.

The *badness* β of the breakpoint is defined as (here $\lfloor x \rfloor$ denotes the integer part of a real number x)

$$
\beta = \begin{cases}
+\infty, & r < -1; \\
\lfloor 100|r|^3 + 0.5 \rfloor, & \text{otherwise.}
\end{cases}
$$

The badness does not depend on the sign of r. TeX never allows more shrink than specified, and therefore the badness is defined to be infinite if $r < -1$. We emphasize that TeX *does* allow more stretch than what is specified, so $r > 1$ is allowed in the second line of the badness calculation above.

It is possible in ConTeXt to show beside each line the badness values that were calculated for each used breakpoint; see Figure 2.6. The turquoise bars at right (grayscaled for print) indicate if the line is set tight or loose.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

[01] b=11 from s=1 b=11
[02] b=62 from s=2 b=62
[03] b=52 from s=3 b=52
[04] b=6 from s=4 b=6
[05] b=0 from s=6 b=0

Figure 2.6   The turquoise bars indicate how much the first, third and fourth lines are stretched and therefore a bit loose. The second line is shrunk and therefore tight.

In Figure 2.7 we have set the paragraph on a narrower width. To the left, with hyphenations enabled, we get a solution that is okay, while to the right the word Calling is sticking out. Here the second run failed and TeX gave up on finding a good solution.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.7   Left: a narrow paragraph, with hyphenation enabled. Right: The same narrow paragraph, with hyphenation disabled.

In TeX, hyphenations are controlled by penalties. We want to avoid hyphenation if possible, but we do not want to disable it completely, since then TeX will sometimes fail to find feasible solutions. Each hyphenated line costs a \hyphenpenalty, and there is a competition between different costs that determine how TeX will break the paragraphs.

Going even narrower, we will eventually end up in a situation where none of the first two passes succeed, with or without hyphenations enabled. With \emergencystretch unset (left in Figure 2.8), we get overfull lines sticking out. We absolutely want to avoid that, if at all possible. To the right we set \emergencystretch to 2em. This means that each line gets an amount of 2em extra stretch to distribute. For this paragraph 2em was enough for TeX to find a solution without any line sticking out.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.8   Left: a narrow paragraph. TeX has failed to find a solution with its both first runs, and we see several overfull lines. Right: The same paragraph with the emergency run enabled by setting \emergencystretch to 2em.

As discussed above, when TeX reads a paragraph, it will put possible breakpoints in a tree-like structure (in fact a single-linked list, with new entries added at the begin-

ning). If the badness is greater than the (pre)tolerance, the breakpoints will be discarded. Once the paragraph is read, TEX will work on the tree with the breakpoints that survived, as shown in Figures ?? and 2.3 (unless there was a complete failure, and then TEX will typically produce a paragraph where some line sticks out, as in Figure 2.8, left). To choose among the possible breakpoints, TEX calculates demerit values for each possible solution, and the final choice is the path in the tree that adds up to the least total amount of demerits.

Let us explain the content in Figure ??. The first column denotes what linebreak we are looking at, so for example the three first rows all correspond to the breakpoint after the first line. Since there are three such rows there are exactly three possible line breaks. The second column is an index for the possible breakpoints. The third column shows what breakpoint each breakpoint is pointing at. We see that the first three point at 0, which means the beginning of the paragraph. The fourth breakpoint is pointing at breakpoint 1. The values in the fourth column are the badness values. The fifth column keeps the (accumulated) demerits. The sixth column shows the fitness class of the break (more on that below) and the seventh column shows the type of break. The breakpoints that have colored numbers are the ones that are used.

After that we get a summary on what paths in the tree were still valid at the end; here there were seven, and $15 \to 8 \to 4 \to 1$, marked in color, was the chosen one. Looking at the paragraph we note that the breakpoints 15 and 16 sit at the same place. The difference is that they point to different previous breakpoints (8 and 7, respectively).

In the summary we also see what subpass was used (P means the pretolerance run). We also see that the total demerits was 1139, that we did not use (extra) paragraph passes, and not looseness (more on that later).

Next, we describe how the demerits are calculated. For each breakpoint, the following happens. Let β be the badness, ℓ the line penalty (by default set to 10), π the possible penalty, and α the additional demerits that correspond to a certain breakpoint (but that usually comes from a combination with the previous one). Then the demerits value δ for that breakpoint is defined by

$$
\delta = \begin{cases}
(\ell + \beta)^2 + \pi^2 + \alpha, & \text{if } \pi \geq 0; \\
(\ell + \beta)^2 - \pi^2 + \alpha, & \text{if } -\infty < \pi < 0; \\
(\ell + \beta)^2 + \alpha, & \text{if } \pi = -\infty.
\end{cases}
$$

The penalty π can come, for example, from a hyphenation (\hyphenpenalty is often set to 50) or a break inside a formula. The only places where traditional TEX breaks inside formulas are after binary operators such as + (default penalty 700) and binary

relations such as = (500). This means that hyphenations are preferred over breaks inside formulas.

The additional demerits, α in the formula, come from the interplay of neighboring lines. There is for example `\doublehyphendemerits` (10000) that gets added when consecutive hyphenated lines are considered during line breaking, `\finalhyphen-demerits` (5000) that is added if the final breakpoint of the paragraph is hyphenated and also `\adjdemerits` (10000) that is added when consecutive lines are considered to be incompatible with each other, say one with fitness class tight and one with loose. The values given above in parentheses are the ones that Knuth set up for plain TeX; they have survived also in other macro packages.

We stress that in the formula for demerits, the $\ell + \beta$ and π are squared, while α is not. This means that $\alpha = 10000$ corresponds to a penalty $\pi = 100$. This is important to have in mind when setting up the parameters, since when TeX chooses the line breaks, these are the kind of terms that compete with each other, and that influence the final choice.

Traditional TeX has four fitness classes: tight, decent, loose and very loose. They are attached to breakpoints and depend on the badness. The adjacent demerits are added when we jump over at least one fitness class. Looking at Figure 2.9, we see that it happens when we go from A to either C or D (from tight to either loose or very loose), or when we go from B to D (decent to very loose). The same amount is added both for jumps from A to C and from A to D, even though the latter is likely worse.

Problematic paragraphs can be tweaked manually. We can locally increase the tolerance to make TeX accept less good solutions, we can enable font expansion in order to stretch or shrink characters slightly, and thus enable line breaks that otherwise would be considered bad. Excessive use of expansion might lead to visually incompatible lines and ugly results. We can also, as mentioned and shown above, set the `\emergencystretch` to a positive value, and hope for a good final run.

Sometimes, the last line of a paragraph is too short. It can look bad if indentation is enabled and the size of the indentation is approximately the same, or even bigger, than the width of the last line in the paragraph before. One way to avoid such short last lines is to use orphan penalties (we will come back to them). If set to 10000, TeX may no longer break between the last two words.

When optimizing for the number of lines to stay on a page, or to add one extra, it is sometimes possible to shorten (or lengthen, but that usually does not give good results) paragraphs with help of `\looseness`. As an example, look at Figure 2.10. By adding `\looseness-1` we ask for a paragraph that is one line shorter than the
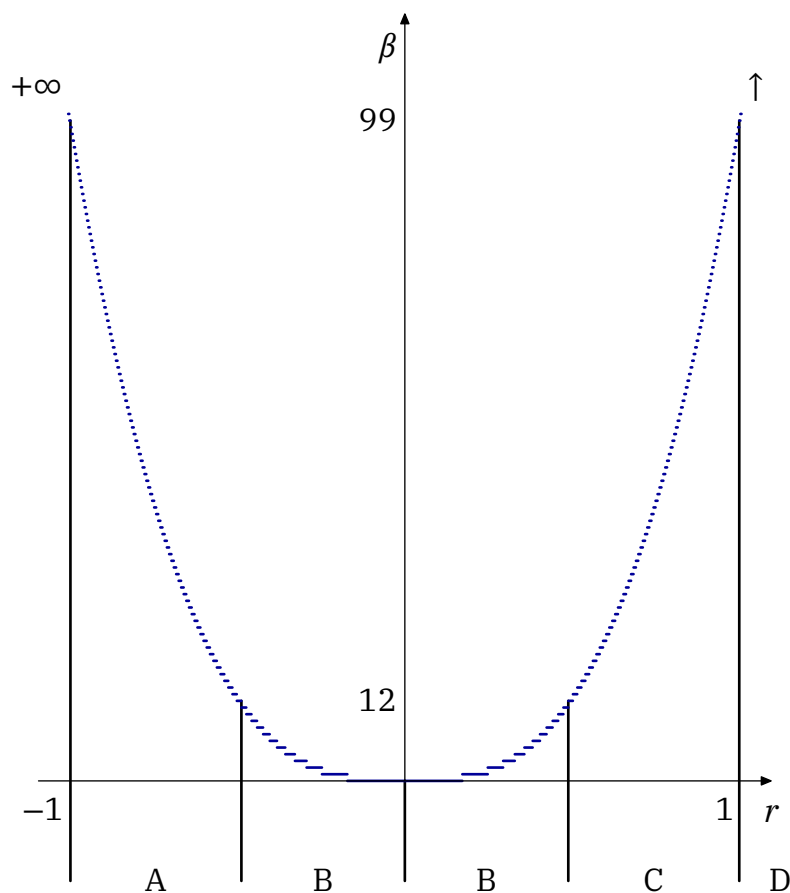
Figure 2.9   Traditional fitness classes. A: tight, B: decent, C: loose and D: very loose. The graph shows the badness β as a function of the adjustment ratio r. The jumps are due to the integer part in the formula (we want integers). The division into fitness classes is done so that the subintervals on the r-axis have the same length. This gives the thresholds 12 and 99 for the badness.

number of lines that the optimal paragraph considered by T_EX has, if possible. In this case it succeeded.

We emphasize once more that this will only work if there is a case among the possible solutions that T_EX has collected that has the number of lines asked for. Also, an otherwise-successful pretolerance run might be discarded in the hunt for a paragraph with the number of lines asked for. In LuaMetaT_EX the user will get a message in the log that tells if it was successful or not. The `\looseness-1` is local, bound to the current paragraph, so it is completely a manual tweak.

2.3 Introducing paragraph passes

We are now ready to discuss the LuaMetaT_EX extension of the traditional paragraph builder, where we set up and use our own paragraph passes (par passes for short). The idea is that more runs on each paragraph, and the possibility of configuring the

I can give a good example of this procedure. At one time, in 1927, I was
playing around with three 2×2 matrices whose squares are equal to unity
and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed
that if one multiplied them into the three components of a momentum so
as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was
just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make
of it?

A paragraph with a short last line.

I can give a good example of this procedure. At one time, in 1927, I was
playing around with three 2×2 matrices whose squares are equal to unity
and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed
that if one multiplied them into the three components of a momentum so as
to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just
$p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

By using `\looseness-1`, the paragraph is shortened by one line.

Figure 2.10   A paragraph with a short last line.

relevant parameters for each run independently, will give a higher quality in gen-
eral. The model with badness, penalties and demerits is kept, and we also keep the
same logic when running over the paragraph to decide which breakpoints to keep
and which to throw away. We did in fact test altering the formulas both for badness
(for instance, why the cube?) and demerits, but we did not see any improvements.

Let us introduce the new concept by studying examples, where we step-wise show
what we can do and the different parameters available. We will use some low-level
setups here. Later we will indicate a few possible high-level interfaces available to
ConTEXt users. We start with a very simple example where we use two passes, the
first with tolerance set to 50 and the second with tolerance set to 100. We disable
hyphenation.

```
\parpasses 2
  hyphenation    0
  tolerance     50
next
  tolerance    100
\relax
```

Here, the `\parpasses 2` specifies that we will use two passes. The keyword `next` is
the divider for successive par pass setups. Values are inherited, so hyphenation is
still disabled in the second run. The specification ends with `\relax` just to be sure
that we do not read on. We can enable these par passes with `\linebreakpasses 1`,
and have a look at the test paragraph.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.11   With the same text width as in Figure 2.2 it happens that we get the same breakpoints with the par passes enabled.

This happens to work out well; we get the same result as the traditional parbuilder gave us. If we go a bit narrower, we will get an overfull line pretty soon, since the tolerance is low and we do not hyphenate: see Figure 2.12.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.12   Hyphenation is disabled in the defined par pass. A narrower width leads to a problematic paragraph.

One way to prevent the overfull line could be to enable hyphenation in the second run; see Figure 2.13. Except for the lower tolerance, this is close to the traditional TeX setup that we started with, with a pretolerance run and a tolerance run, only a bit stricter.

```
\parpasses 2
  hyphenation    0
  tolerance     50
next
  hyphenation    1
  tolerance    100
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.13   Hyphenation enabled in the second run.

In the example above, we specified the values of a few parameters. Then the logic follows the traditional paragraph builder: TeX does a run with the settings of the first pass. If it is successful, we are done. If not, it will run the second pass, and since we have no more passes, it is marked as a final pass. This means that TeX will make sure

that something is returned, even if it fails to fulfill the constraints of the parameter values. That is why we got an overfull line in Figure 2.12, with hyphenations disabled. Another option for fixing the overfull line would be to increase the tolerance. A value of 200 would work in this case.

2.4 Hyphenation

We just saw how one can turn hyphenations on and off in par passes. The outer value of `\hyphenpenalty` (50) will be used; for technical reasons, it cannot be changed inside the par passes. Therefore, we have introduced the keyword `extrahyphenpenalty`, that adds to the exterior `\hyphenpenalty`.

```
\parpasses 2
  hyphenation          0
  tolerance           50
next
  hyphenation          1
  tolerance          100
  extrahyphenpenalty 150
\relax
```

Here we have set `extrahyphenpenalty` to 150. It is additive, so with `\hyphenpenalty` set to 50, the total penalty for breaking at hyphens becomes 200. The outcome would in this case be the same, whatever finite value of `extrahyphenpenalty` we give, because this is essentially the only solution that is available. We would need an extremely high value (9950) to prevent the hyphenated line, but that would just mean that we forbid hyphenations, so we could then equally well not enable it. And we saw that the paragraph did not come out well without hyphenation.

It is also possible to influence hyphenations by setting the parameters `doublehyphendemerits` and `finalhyphendemerits`. We need to remember that these are indeed demerits, and therefore of the order of penalties squared.

2.5 Font expansion

It has become very popular in so-called microtypography to use font expansion, i.e. to stretch and shrink glyphs just slightly. This can reduce the amount of hyphenations needed, and it can also even out the spacing a bit, leading to better paragraphs. An excessive use of expansion quickly becomes ugly, as we can see in many newspapers, with narrow columns.

One of the nice aspects of par passes is that one can apply expansion selectively. It is indeed possible to enable and disable expansion. Below we disable it in the first

run (it might have been enabled outside the par passes with `\setupalign[hz]`) and then enable it in the third run by doing `adjustspacing 3` to expand glyphs and font kerns. We set the step to 1, maximum shrink to 10 (that means 1%) and maximum stretch to 15 (that means 1.5%). These values might seem small, but, by using several paragraph passes, one can increase the values in the latter passes, and thereby not use more than needed.

```
\parpasses 3
    tolerance             50
    hyphenation            0
    adjustspacing          0
next
    tolerance            100
next
    adjustspacing          3
    adjustspacingstep      1
    adjustspacingshrink   10
    adjustspacingstretch  15
\relax
```

Note that we disabled hyphenation in this setup. The narrow paragraph, that before introduced the lines sticking out, now typesets okay; see Figure 2.14. We get a line-break inside a formula, and we will soon come back to that problem.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.14   The tight paragraph, with expansion. The blue and red numbers indicate the amount of stretch and glue, respectively.

In the paragraph in Figure 2.14 we used the command `\showmakeup[expansion]` to show the amount of stretch and shrink for each character. In ConTEXt it makes sense to set `expansion=quality` as a font feature. This will take the difference in characters into account when spreading the stretch and shrink.

```
\definefontfeature
  [default]
  [default]
  [expansion=quality]
```

For those who are familiar with expansion in the other engines we remark that in LuaMetaTEX we implemented it a bit differently. For instance, we have an expansion

as well as compression factor per glyph. Instead of 'freezing' the step, stretch and shrink in a font definition, we can change it any time. Sensible values can still be bound to a specific font switch but when we set the adjustment properties in a pass those values are taken instead. Moreover, instead of initializing the glyph compression and expansion factors when a font is loaded we (can) delay this until it is needed. Experiments demonstrated that it is less often needed that one might think, so not all fonts need this to be set up. For this reason, font expansion in the par passes has only a small impact on run time.

2.6 Mathematics

When developing this extension we were also busy with extending math support in the engine and as a consequence we took math into account. For instance, we have a parameter that can influence the inter-atom penalties.

```
mathpenaltyfactor 500
```

This reduces several math penalties by 50%. To minimize the number of breaks inside math, we can start out with a large `mathpenaltyfactor` in the first run, and decrease it during later runs. We consider the narrow paragraph, but under more natural tolerance values, and without hyphenation (Figure 2.15).

```
\parpasses 2
    tolerance    100
    hyphenation    0
next
    tolerance    200
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.15   A narrow paragraph. We get penalties before short formulas and after binary operators and binary relations. In this case the penalty does not prevent a break in a formula.

We get the default penalty of 700 after binary operators (the plus signs). We would also get 500 after binary relations, if we had any. We do also get the ConT_EXt specific penalties of 150 before short formulas. We do indeed get a line break inside the formula. If we want a run that prohibits both the breaks after the plus signs and before

the short formulas we need to multiply by a factor that ensures that both are 10000 or more. In this case 67 is sufficient; see Figure 2.16.

```
\parpasses 2
    tolerance          100
    hyphenation          0
    mathpenaltyfactor 67000
next
    tolerance          200
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.16   With high enough `mathpenaltyfactor`, we can forbid TeX to break inside formulas and before short formulas. In this case it was not successful.

This was not especially successful, since we kept the high value through all paragraph passes. We reiterate that it might be better to forbid those breaks in the first pass(es) and then maybe decrease the factor for later runs.

The `mathpenaltyfactor` also works in combination with forward and backward penalties, which can be used to try to avoid line breaks in the beginning or at the end of a longer inline math formula. A possible setup for these is given below.

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

These will add a penalty of 200 to the first and last available breakpoints in an inline math formula, and a penalty of 100 to the second and second from last.

2.7 We have an emergency!!

Oh, just kidding! The word emergency in the traditional TeX primitive `\emergen-cystretch` might have been a bit unfortunate, since it is not a bad idea to enable it, sparingly of course.

If we set the emergency stretch to `2em` in the example with low tolerance, we do indeed get the break inside the formula (Figure 2.17). With emergency stretch set to `1em` above, it won't help (Figure 2.18).

```
\parpasses 2
```

```
  hyphenation          0
  tolerance           50
next
  tolerance          100
  emergencystretch    2em
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.17   Paragraph set with `\emergencystretch=2em`.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.18   Paragraph set with `\emergencystretch=1em`.

There are more "cheats". In Figure 2.19 we use an emergency stretch of 1em and also mess with the width of the paragraph, to the right. The 20 here means 2%.

```
\parpasses 2
  hyphenation              0
  tolerance               50
next
  tolerance              100
  emergencystretch        1em
  emergencyleftextra      0
  emergencyrightextra    20
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.19   Paragraph set with `emergencyrightextra`.

Another one is `emergencywidthextra`: use a different width when the line breaks are decided, but not apply it in the end. This means it only works out well if lines

have stretch and shrink. In the example in Figure 2.20 we use 2% extra width. This should probably only be used in true emergencies, if at all.

```
\parpasses 2
  hyphenation           0
  tolerance            50
next
  tolerance           100
  emergencystretch     1em
  emergencywidthextra  20
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.20   Paragraph set with `emergencywidthextra`.

We have so far set the emergency stretch explicitly, in terms of font `em` units. If we have hanging indentation or parshapes, the widths of different lines in the paragraph will vary. One can then argue that it makes more sense to set the amount of emergency stretch as a percentage of the line width, even if it does not matter for most paragraphs. In Figure 2.21 we set the stretch to 4% of the line width, which was sufficient this time.

```
\parpasses 2
  hyphenation           0
  tolerance            50
next
  tolerance           100
  emergencypercentage  40
\relax
```

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.21   Paragraph set with `emergencypercentage`.

It might happen that \emergencystretch is set to a positive value outside of the par pass setups (for example via \setupalign[stretch]). When we go in to the

par passes, we can use `emergencyfactor` to handle that. We can start by setting it to 0 in the first pass to be sure to disable the emergency stretch, and then update it to a positive value in a later run to enable it.

```
\parpasses 2
  hyphenation        0
  emergencyfactor    0
  tolerance         50
next
  tolerance        100
  emergencyfactor 1000
\relax
```

2.8 More penalties

An example above showed the `extrahyphenpenalty` parameter, which is specific to paragraph passes. There are a few more penalties available. The ones below can also be set by primitives. An orphan penalty can prevent a line break before the last word in a paragraph (we come back to that one), and a toddler penalty might prevent a line break before a single glyph.

```
linepenalty    100
orphanpenalty  200
toddlerpenalty 200
```

We show one example with `linepenalty`. Earlier we used `\looseness-1` to get the paragraph one line shorter. In Figure 2.22 we succeed in obtaining the same paragraph by increasing the `linepenalty` from 10 to 100. It is, however, difficult to predict when it will work.

```
\parpasses 2
    tolerance     50
    hyphenation    0
    linepenalty 200
next
    tolerance    100
\relax
```

It's worth mentioning that the first versions of TeX did not come with the `\linepenalty` parameter. The corresponding number was then 1 instead of the 10, which is probably used everywhere now.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.22   Shortening a paragraph with a higher `\linepenalty`.

The orphan penalties can be problematic if set too aggressively, in particular for short paragraphs that often occur in novels with a lot of dialogue. In Figure 2.23 we see such a problematic example, where we have prohibited breaks before the last word by setting the penalty there to 10000.

```
\parpasses 1
    tolerance        100
    orphanpenalties 1 10000
\relax
```

This is just a short sentence that is just a bit longer than one line.

Figure 2.23   A one-liner with too-strict orphan penalties.

To avoid this problem we have factors that can be used. Below we multiply by 0.1 if the paragraph has one line break, 0.5 if it has two and 1.0 if it has more than two. We see in Figure 2.24 that this is sufficient; we can now break before the last word.

```
\parpasses 1
    tolerance         100
    orphanpenalties   1 10000
    orphanlinefactors 3 100 500 1000
\relax
```

This is just a short sentence that is just a bit longer than one line.

Figure 2.24   A one liner with strict
orphan penalties and multipliers.

Let us also show an example where we set toddler penalties both to the left and the right. If you are able to zoom in Figure 2.25, you will see that we get penalties of 50 sitting to the right of the single character letters, and 25 to the left of the leftmost one. The `\parfillrightskip` was set to 0pt here, to get a bit extra space between the words so that the penalties show better. We do not know if there are languages where single-letter words can be stacked like this.

```
\parpasses 1
    tolerance        100
```

```
    toddlerpenalties 1 options 2 50 25
\relax
```

Some write: I owe you one.
traditional:subpass 01

The kids write: I o u 1.
traditional:subpass 01

Figure 2.25   Penalized toddlers.

2.9 Being more granular

It is possible to specify the number of fitness classes to be used. We saw before that traditional TeX uses four: tight, decent, loose and very loose. By invoking

```
\setupalign[granular]
```

we enable more. You can see in Figure 2.26 that they are evenly spread out regarding the adjustment ratio, and how they are related to the badness values. This should be compared with Figure 2.9.
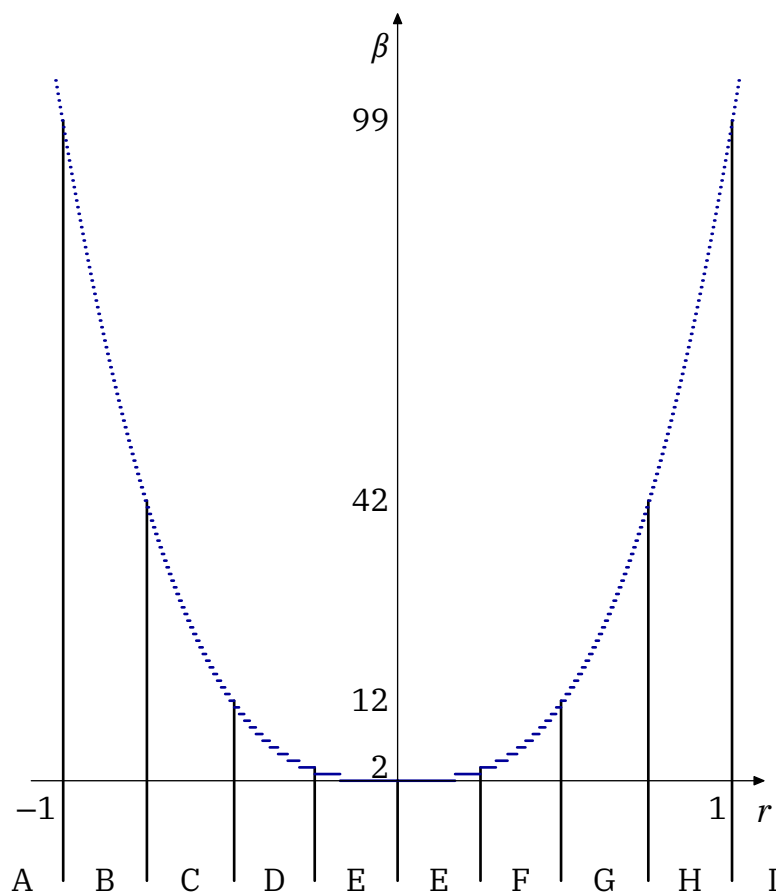
Figure 2.26   Granular fitness classes. A: very tight. B: tight. C: almost tight. D: barely tight. E: decent. F: barely loose. G: almost loose. H: loose. I: very loose.

The granular fitness classes are defined by a `\specificationdef` command (more about them later). The classes are defined to spread evenly over the adjustment ratios, just as in the non-granular situation.

```
\permanent \specificationdef \granularfitnessclasses
  \fitnessclasses 9
  99
  42 % .75
  12 % .50
   2 % .25
   0 % .00
   2 % .25
  12 % .50
  42 % .75
  99
```

It becomes more meaningful to enable the granular mode if we also configure how these fitness classes are to be used. As previously mentioned, in traditional TeX the `\adjdemerits` is added whenever we jump over at least one fitness class when going from one line to the next. We can use `adjacentdemerits` in the par passes. For example,

```
adjacentdemerits 4 0 5000 7500 10000
```

defines four levels of adjacent demerits. For two consecutive linebreaks with neighboring fitness classes, no demerits is added. If we jump one step 5000 is added, jumping two steps cost 7500 and three steps (or more) cost 10000.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them $\sigma_1, \sigma_2, \sigma_3$, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.27 The test paragraph, set with more granular fitness classes.

1	1	0	13	5529	almostloose	glue		10	6	0	40429	decent	glue	14	741
	2	0	3	490169	barelytight	penalty	4	11	8	14	11515	almostloose	glue	15	1063
	3	0	45	10525	tight	math		12	7	14	6242	decent	glue	pass : 4 demerits : 6139
2	4	1	7	5818	barelyloose	glue		13	8	8	6039	decent	glue	subpass : 1 looseness : 0
	5	3	6	43281	barelyloose	glue		14	7	8	15011	almosttight	glue	subpasses : 2
	6	3	38	35329	almosttight	glue		15	10	1	40550	decent	glue
3	7	4	8	6142	barelyloose	glue	11	841
	8	4	1	5939	decent	glue	12	741
	9	5	15	43906	almostloose	glue	13	841

Figure 2.28 Information on the breakpoints that TeX used for the paragraph in Figure 2.27.

In Figure 2.28 we see fitness classes that we did not see before, such as barely tight and almost loose. We see in Figure 2.29 that the tree is slightly different from the one in Figure 2.3.
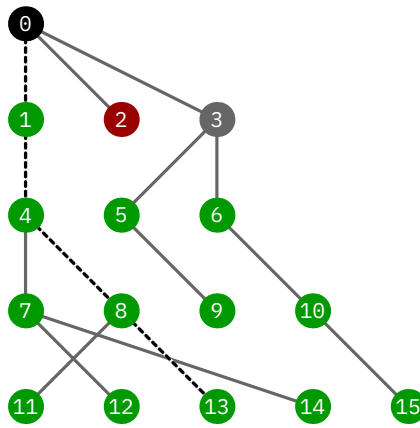
Figure 2.29   The tree corresponding
to the paragraph in Figure 2.27.

We got here the same linebreaks as for the traditional parbuilder (Figure 2.2). But in the traditional case we had fitness classes loose, decent, decent, which means that we paid no demerits for them. Now we got almost loose (cost 5000), barely loose (0), decent (0). Thus, the total demerits in this case landed at 6139 instead of 1139.

Going back to the default fitness classes, one can use

`\fitnessdemerits 0`

and to go granular in only one par pass, one can use

```
...
    fitnessclasses    \granularfitnessclasses
    adjacentdemerits \granularadjacentdemerits
next
    classes           \matchallfitnessclasses
...
```

where the `\granularadjacentdemerits` have been defined to be compatible with the more granular fitness classes. The `classes` parameter (a bitset) tells the builder to check all set classes; the constant is a generous `"FF`.
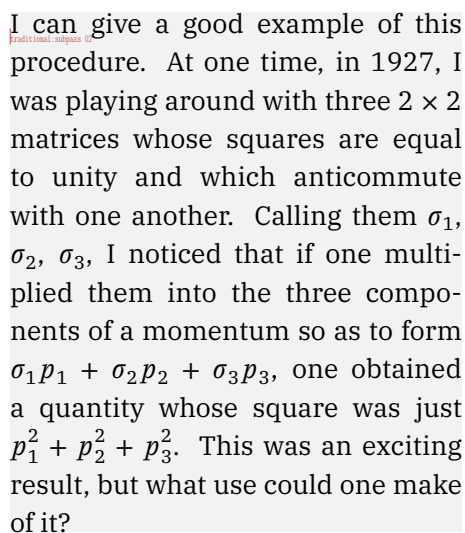
On a bigger project, we have seen only a few changes when enabling the granular setup. Since they are few it is difficult to say something general about quality, but we expect that the neighboring lines are slightly more compatible.

2.10 Other demerits
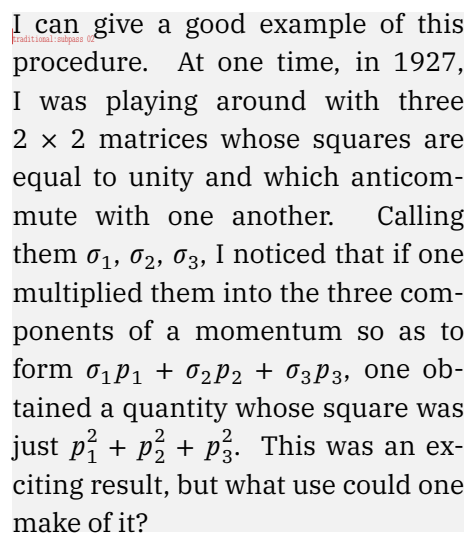
We recall that it is the demerits of the paragraph that TEX uses as a cost function to select the best set of line breaks; the solution with minimal demerits wins. We

emphasize again that the additional demerits are not added to singular breakpoints, but to combinations of breakpoints that fulfill some condition.

We have already seen how the granular fitness classes could be used, together with `adjdemerits` (defaults to 10000), or rather the plural version `adjacentdemerits`, to be able to detect smaller differences in badness values between consecutive lines. There are other demerits we can set. In Figure 2.30, we compare results with `doublehyphendemerits` set to zero (left), and set to a high value (right), preventing two consecutive lines from being hyphenated.

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

I can give a good example of this procedure. At one time, in 1927, I was playing around with three 2×2 matrices whose squares are equal to unity and which anticommute with one another. Calling them σ_1, σ_2, σ_3, I noticed that if one multiplied them into the three components of a momentum so as to form $\sigma_1 p_1 + \sigma_2 p_2 + \sigma_3 p_3$, one obtained a quantity whose square was just $p_1^2 + p_2^2 + p_3^2$. This was an exciting result, but what use could one make of it?

Figure 2.30   Left: A narrow paragraph set with `doublehyphendemerits` set to 0. Right: The same paragraph with `doublehyphendemerits` set to 300000.

We also have `finalhyphendemerits` that can be used to discourage the last breakpoint from being hyphenated. Its default value is 5000.

The settings are equivalent to the primitives and more about them can be found in regular TeX documentation. We also have twin demerits:

```
lefttwindemerits  2000
righttwindemerits 2000
```

These discourage line breaks where words at the beginning or end of lines are the same; be aware that this doesn't prevent mid-line occurrences. And of course it puts more constraints on the solution and has to work with other constraints. More about this feature can be found in another recent TUGboat article.[3]

---

[3] "Twin demerits", Hans Hagen and Mikael P. Sundqvist, TugBoat, vol. 45, no. 2, pp. 362–369, 2024, https://tug.org/TUGboat/tb45-3/tb141hagen-twins.pdf.

2.11 Conditionally entering par passes

We have seen several examples of using par passes, where the standard logic of TEX is kept: if TEX is happy after a run, we are done with the line breaking. It is possible to also enter par passes conditionally. There are three main criteria that we can use. The valid criteria keys are demerits, threshold and class:

- demerits: the overall measure that TEX uses to select the best choice.
- threshold: over- or underfull lines
- classes: compatibility between successive lines.

The first is not that useful because it is hard to come up with some good numbers. Longer paragraphs typically have higher demerits than shorter, and for very long paragraphs some shortcuts are taken and large values get clipped in order not to overflow numbers. We will discuss the classes option soon.

In the traditional paragraph builder it is difficult to go back and deduce what decisions were made during the runs, and how and why they were made. The values are not kept, except for the demerits, but those are recalculated as we go.

When we add more passes we don't know in advance what is the final pass, but we need one because in the end we must have a result. We could of course always add a final one automatically but then we might just as well take the last one anyway. The multiple pass mechanism will always do the regular pretolerance and tolerance passes but we can set the values in a par pass definition. We have two situations:

1. When we have a criterion in the first par pass, we will first do the two tolerant passes. The second tolerant pass is a final pass so we do have a result but we check for further actions in the list of par passes.
2. When we have three or more par passes, the first two will act like tolerance passes when there are no criteria. Again the second one is a final pass that can have a follow up.

Here we have a single par pass of the first category:

```
\parpasses 1
  threshold          0.025pt
  tolerance          300
  emergencyfactor 1000
\relax
```

and here is an example of the second:

```
\parpasses 3
```

```
  tolerance        100
next
  tolerance        200
next
  threshold          0.025pt
  tolerance        300
  emergencyfactor 1000
\relax
```

The second one has no criteria, so the last pass becomes the final pass, which kicks in when none of the previous ones gave an acceptable solution.

The `classes` key is the most difficult one to describe. In ConTeXt we can add

```
\parpasses 1
  classes                 \indecentparpassclasses
  tolerance        300
  emergencyfactor 1000
\relax
```

and that needs an explanation. When TeX looks at lines it will use adjacent demerits to penalize neighboring lines that are space wise incompatible. The `\indecent-parpassclasses` condition will let you enter the par pass if there are any lines that ar flagged as not being decent.

In ConTeXt, when using the granular mode described above, we have these constants defined:

```
\integerdef\verylooseparpassclass    "0001
\integerdef\looseparpassclass        "0002
\integerdef\almostlooseparpassclass "0004
\integerdef\barelylooseparpassclass "0008
\integerdef\decentparpassclass       "0010
\integerdef\barelytightparpassclass "0020
\integerdef\almosttightparpassclass "0040
\integerdef\tightparpassclass        "0080
\integerdef\verytightparpassclass    "0100

\integerdef\allparpassclasses        "FFFF
```

The definition of `\indecentparpassclasses` is then:

```
\integerdef\indecentparpassclasses\numexpr
```

```
    \allparpassclasses
  - \decentparpassclass
\relax
```

As you see the condition is really using a bitset, but it is easier to have names for them. There are a few others predefined:

```
\almostdecentparpassclasses
\looseparpassclasses
\tightparpassclasses
```

and you can define your own just as we showed above.

In addition to the `demerits`, `threshold` and `classes` criteria mentioned above, we can also decide if entering (using) a par pass with the following keys

- `ifadjustspacing`: enter if `expansion` is enabled.
- `ifemergencystretch`: enter if `emergencystretch` is enabled.
- `ifglue`: enter if there is anything to stretch or shrink.
- `iftext`: enter if the paragraph has text (glyphs/discretionaries).
- `ifmath`: enter if the paragraph has math.
- `unlessmath`: enter if the paragraph does not have math.

A new block of parameters is marked by `next`. With `quit` processing passes can be stopped, and `skip` will bypass a pass. These last two are mostly for testing.

To sum up, we have three situations:

- traditional mode, up to three passes,
- mixed mode, first two traditional passes and then additional ones,
- par passes that likely include traditional setups,

and we have various different ways to condition on entering the par passes.

2.12 A bit of infrastructure

For administrative purposes we have the directives `callback`, `identifier`, and `linebreakchecks`, as well as `linebreakoptional` to select what optional content to enable.

We can add an identifier:

```
\parpasses 3
  identifier        1
```

```
  tolerance        100
next
  tolerance        200
  hyphenation        1
next ifemergencystretch
  emergencyfactor 1000
\relax
```

This identifier will be used in reporting and in ConTEXt we can relate this to a more meaningful name, like 'default'. We can avoid altering the current par pass by defining an alias:

```
\specificationdef \parpassdefault \parpasses 3
  identifier         1
  tolerance        100
next
  tolerance        200
  hyphenation        1
next ifemergencystretch
  emergencyfactor 1000
\relax
```

We use a generic `\specificationdef` and by just issuing the given name the par pass is activated. However, one also has to set `\linebreakpasses` to a positive value to let it do its work.

2.13 Changing par passes locally

We saw how to use `\looseness` to manually (try to) tweak a single paragraph in the traditional par builder. We have a similar local par pass mechanism. With `\parpassesexception` we can locally use a specified par pass setup for the current paragraph. It has to be called just before the paragraph in question, as in:

```
\parpassesexception \mylocalparpasses
Paragraph comes here ...
```

will use the par passes setup `\mylocalparpasses`, which must have been previously defined with a `\specificationdef`. This opens it up for a simple but complete local control when needed.

2.14 After breaking the paragraph into lines

Breaking a paragraph into lines and, at some asynchonous, point breaking pages are separate processes. The first process has related penalties and demerits that are part

of the decision making that are no longer relevant once the work is done. The second process also has penalties to consider, for instance widow and club penalties. These are inserted between lines by the par builder because it is that routine that, after optimal breakpoints have been determined calls out to a post line break routine that constructs the lines. The lines themselves as well as various glue and penalties, plus possible `\vadjust` and `\insert` material, are added to a current list of contributions that is eventually transferred to the page. So, it makes sense to mention them here.

It goes unnoticed, but the broken line is in practice no more than a begin and end point in the horizontal list that enters the routine. Every line is just a range and although the decisions were made using glue and optionally font expansion, the original nodes are still there. So, when that range has to become a line, the horizontal pack routine is called to wrap it into a `\hbox`, and, as with any horizontal box construction, it will recalculate what the final glue will be and what expansion is applied, based on what the par builder decided. Also, before packaging, the left and right skip, indentation, paragraph related shape measure etc. are injected. This somewhat redundant effort is fast enough not to be of impact.

An important activity in this packing is that we (when enabled) can normalize the line. Depending on what line we are, we have a lot of skips to consider (for practical purposes items that are actually kerns, like indentation, also use glue nodes):

```
leftskip lefthangskip leftparskip leftinitskip
indentskip [content] correctionskip
rightinitskip rightparskip righthangskip rightskip
```

We also make sure that direction nodes are balanced and math is well indicated across lines. Discussing this process is beyond what this article focuses on, but you can imagine that it involves some code. This pays off in nicer code at the Lua end when we want to mess with the lines afterwards.

The abovementioned widow and club penalties (plus some more) are taken from the singular and plural commands `\widowpenalty`, `\widowpenalties`, `\clubpenalty`, `\clubpenalties`. In LuaMetaTEX these values are stored in the initial par node.

For the record: there are uet more penalties that matter, for instance we have `\shapingpenalty` that can prevent breaks in a parshape or hanging setup and `\singlelinepenalty` that penalizes a two line result. We don't discuss how display math is handed in line breaks and wrapping up, but just mention that we can have a display formula that combines with the previous and upcoming paragraph. In that case the builder sees three paragraphs as one and display math as three lines, which of course influences what is seen as the current line width when shaping the whole.

It doesn't affect the discussed break mechanism. In ConT_EXt we handle display math differently, so we have not added features for mixed-in display math.

A par pass definition is what (in LuaMetaT_EX) we internally call a specification command. Other examples of specification commands are `\parshape` and the mentioned plural penalties. Each of their values is a pointer to a node, and because the amount of data can differ these have a variable size. In LuaT_EX they are taken from the regular pool and when the set of values change another sized one is needed. Released nodes are kept in a pool but one can think of scenarios where too many different sizes will create a bit of a mess. This is why in LuaMetaT_EX we allocate the variable part dynamically as an independent 'array' of parameters.

The reason for mentioning these details is that, because of the decoupling between the handful of primitives and the way their information is stored, it started making sense to provide ways to create variables as with registers. This brings us to an example:

```
\widowpenalties 4 2000 500 250 0
```

We can also say:

```
\specificationdef \lesswidowpenalties \widowpenalties
  4 2000 500 250 0
\relax
```

and then use `\lesswidowpenalties` to enable this set of penalties.

The `\specificationdef` command can also be used to define par passes, as we saw above. Using these definitions is not only faster but also has the advantage that we can provide interfaces in ConT_EXt in the way we like. It also makes it easier to reset the plural penalties to default values. An even more important feature is that we can get rid of the singulars which is a big benefit because of the way the engine works. When ε-T_EX introduced these plurals it had to remain compatible so this is what happens there:

- When `\widowpenalties` is set, `\widowpenalty` is ignored.
- When `\widowpenalties` is reset, `\widowpenalty` kicks in again.

Resetting `\widowpenalties` is done with:

```
\widowpenalties 0
```

That said, what about the following?

```
\def\widowpenalty{\widowpenalties 1 }
```

This is very close to what we want but because the last value is used for all that follow we would need this to be compatible.

```
\widowpenalty     500
\widowpenalties 2 500 0
```

This is why we end up with:

```
\permanent\protected\untraced\def\widowpenalty
  {\widowpenalties\minusone}
```

where the negative one sets an option to not reuse the last value. The prefixes declare that the command can't be redefined when overload protection is enabled, that it doesn't expand (in e.g. an `\edef`) and that in tracing it gets reported without its meaning, so basically the users see a primitive. The advantage of this approach is that we only have to deal with one variable. Of course users should be aware of this but few will set these plurals explicitly, leaving that to ConTEXt.

The plural widow and club penalties can result in better results but also add constraints. This means that we can get less full pages or when we have stretch in the white space (if present) possibly inconsistent spacing. We can handle this by limiting the stretch in the vertical spacing combined with overall vertical scaling that we call vz, analogous to hz (Hermann Zapf's initials for expansion); it was Hermann who suggested to us to play with this because "No reader will notice a few percent vertical scaling of the page". Limiting stretch is an engine feature (in the page builder) while vertical scaling is a ConTEXt trick. Applied to the large test document this also helps to make it look great.

Another new feature is that when a paragraph is eventually broken across a page, you might want to distinguish between a left and right page of a spread. It is therefore possible to do this:

```
\widowpenalties 3 options \numexpr8 + 4\relax % largest + double
  5000 7500
   250  500
     0    0
\relax
```

This says: use higher penalties for the right page and when you overlap with club penalties use the larger of the widow and club penalty, i.e., we do not want to add them. The `options` is a bitset that differs per specification.

The `\adjdemerits` parameter controls what demerits get added to lines that have a distance of more than one step in the fitness sequence tight, decent, loose, very loose.

37 A new take on paragraphs

In LuaMetaTeX we have more control over this; for instance we can, as we have seen, have more steps. In that case we also apply different demerits for every distance and even have accumulated demerits. This is controlled by `\adjacentdemerits` and we can redefine the traditional parameter like this:

```
\permanent\protected\untraced\def\adjdemerits
  {\adjacentdemerits\minusone}
```

So, to summarize this part: setting up and using par passes to get better results is worth the effort, but part of this often also involves making sure the vertical penalties are right. This is bound (applied) to the result of line breaking.

2.15 Tracing and debugging

It would be impossible for us to develop these new features without extensive testing, and the testing would be very difficult to do without tracing. There are two ways to trace what the engine is doing: built-in (hard-coded in the engine) reporting, and ConTeXt trackers that use Lua to add visual or report textual information. The first one is probably not that useful unless you need to know what goes on deep inside; the second can help you improve a specific document setup.

When `\tracingpenalties` is set to 1, you will get reports like this, where `l` and `r` refer to the left and right page of a spread where the values kick in when the page is broken in a double sided layout:

```
[linebreak: interline penalty, line 1, index 1, delta 101, total 101]
[linebreak: club l penalty, line 1, index 1, delta 100, total 201]
[linebreak: club r penalty, line 1, index 1, delta 100, total 201]
[linebreak: interline penalty, line 2, index 2, delta 101, total 101]
[linebreak: interline penalty, line 3, index 3, delta 101, total 101]
[linebreak: interline penalty, line 4, index 4, delta 101, total 101]
[linebreak: interline penalty, line 5, index 5, delta 101, total 101]
[linebreak: interline penalty, line 6, index 6, delta 101, total 101]
[linebreak: interline penalty, line 7, index 7, delta 101, total 101]
[linebreak: interline penalty, line 8, index 8, delta 101, total 101]
[linebreak: widow l penalty, line 8, index 1, delta 101, total 202]
[linebreak: widow r penalty, line 8, index 1, delta 101, total 202]
```

When the value is set to 2, you will also get lines that report the `\shaping-penaltiesmode` value that was applied. This is a bitset that determines what penalties will be applied when we have a hanging situation.

```
[linebreak: penalty, line 1, best line 10, prevgraf 0, mode "FF (i=1 c=4 w=2
                                                                      b=8)]
```

Another tracing option is the traditional TeX `\tracingparagraphs` that reports a
lot and even more when its value exceeds 1. Probably more interesting is `\trac-ingpasses`, which reports the parameters used, and, when set to more than 1, also
reports details over the decisions made. We mention also `\tracingtoddlers` and
`\tracingorphans` that might come in handy.

When we discussed and tested these extensions with ConTeXt users, there was some
confusion about `\looseness`. These parameters can, as we have explained, be used
to increase or decrease the number of lines relative to the optimum, if possible.
Any change to the involved parameters might spoil the ability to get that extra line.
With `\tracinglooseness` you get some information about the attempts to fit the de-mands. When tracing with the trackers that show all possible breakpoints it quickly
becomes clear that TeX doesn't discard bad solutions as it goes forward but keeps
them around till (at the end of a successful pass) it tries to loosen.

While developing features like these it helps very much to see what we're dealing
with. For instance, TeX distinguishes between spaces between words (that become
glue) and spaces after punctuation (influenced by the space factors). With `\show-makeup[space]` you can show both (Figure 2.31). This example also shows another
feature: space factoring applied after uppercase characters, in this case shown but
not applied. Think of situations like 'D.E. Knuth'. More control over space factors is
part of the optimizations because we have ways to limit the maximum stretch, just
like TeX already limits the shrink.

I can give a good example of this proced
around with three 2 × 2 matrices whose s
commute with one another. Calling them

spaces between words and after punctuation

D.E. Knuth, author of TeX.

spaces after initials and punctuation

I can give, if needed, an example

space factors and stretch

Figure 2.31

Similarly, we can use `\showmakeup[hpenalty]` to see where horizontal penalties
are applied and `\showmakeup[vpenalty]` for vertical penalties; see Figure 2.32.

I can give a good example of this proced
around with three 2×2 matrices whose s
commute with one another. Calling them

I can give a good example of this proced
around with three 2×2 matrices whose s
commute with one another. Calling them

Figure 2.32

Hyphenation results in injected discretionary nodes; `\showmakeup [discretionary]` lets us see them. The ones at ends of lines eventually get replaced by the content of pre and post fields but we can show the places where they were seen in the rest. We can show them because in LuaMetaTeX we keep track of such decisions in the glyph nodes so we know at what places hyphenation is possible; see Figure 2.33.

I can give a good example of this proced
around with three 2×2 matrices whose s
commute with one another. Calling them

Figure 2.33   We look at all discretionaries, but only longer words get hyphenated.

Expansion is another feature that we might want to track, and `\showmakeup [expansion]` reveals it, see Figure 2.34.

I can give a good example of this procedure
with three 2×2 matrices whose squares a
with one another. Calling them σ_1, σ_2, σ_3,

Figure 2.34   Expansion kicks in.

2.16 Larger example used in a math book

We have experimented a lot with a first year analysis book that Mikael has written with his colleague Tomas Persson, in Swedish. We emphasize that the settings we have ended up using might not fit everyone, but they did seem to work well for this book.

We use five par passes, and we don't enter any of them conditionally; we quit directly if TₑX is happy after a run. Our strategy is trying to avoid both hyphenations and breaking inside of mathematics, as much as possible.

The book contains 3023 paragraphs. A vast majority, 2697 paragraphs, are done by the first run. This is one of the reasons that the extra par passes do not add much overhead.

The first run is a typical pretolerance run. We use no expansion, no emergency stretch, and we accept no hyphenations. Also, the math penalties (inside formulas and before short formulas) are multiplied by 20. This means that they reach at least 10000 and thus such breaks are prohibited.

We follow up with a run with a slightly higher tolerance, and also a very small allowed font expansion, with a stretch of at most 1% and a shrink of at most 0.5%. This one is used 192 times.

In the third run we switch expansion off again, but allow for a tolerance of 300; this is used only five times. In the fourth run we go back to tolerance 200 but increase the possible amount of expansion, and get 76 paragraphs. Finally, in the fifth run we enable hyphenation, but add 200 to its penalty, we increase the amount of font expansion allowed, enable some additional emergency stretch, and also reset the math penalties to the outer values. This run takes care of the 53 remaining paragraphs.

```
\startsetups align:pass:analysisbook
\parpasses 5
  identifier          \parpassdefaultone
  tolerance           100
  adjustspacing         0
  emergencyfactor       0
  hyphenation           0
  mathpenaltyfactor  20000
next
  tolerance           200
  adjustspacing         3
  adjustspacingstep     1
  adjustspacingshrink   5
  adjustspacingstretch 10
next
  tolerance           300
  adjustspacing         0
next
  tolerance           200
```

```
    adjustspacing          3
    adjustspacingshrink    20
    adjustspacingstretch   40
next
    tolerance              400
    hyphenation            1
    extrahyphenpenalty     200
    adjustspacing          3
    adjustspacingshrink    30
    adjustspacingstretch   60
    emergencystretch       1\bodyfontsize
    emergencyfactor        1000
    mathpenaltyfactor      1000
\relax
\stopsetups
```

```
\newinteger\parpassdefaultone
\parpassdefaultone\parpassidentifier{analysisbook}
```

With the last line, we can set up these par passes with

```
\setupalignpass[analysisbook]
```

With

```
\enabletrackers[paragraphs.passes=summary]
```

we get a summary in the log file. Here we find how many times each run was used and also what the paragraphs contained (t is text, d is discretionary and m is math).

```
'subpass 01', count 2697, states 93:t-- 199:td- 1:--m 145:t-m 44:t-- 549:td-
                                135:t-m 1443:tdm 18:td- 69:tdm 1:tdm
'subpass 02', count 0192, states 35:td- 1:t-m 138:tdm 6:td- 12:tdm
'subpass 03', count 0005, states 5:tdm
'subpass 04', count 0076, states 12:td- 57:tdm 2:td- 5:tdm
'subpass 05', count 0053, states 1:t-m 10:td- 40:tdm 1:td- 1:tdm
```

Before comparing some outputs, let us first make clear that it is only a few paragraphs that change. This is good, we do not want to alter TeX's usual very high quality output.

Since we enable hyphenations only in the last run, we get fewer hyphenations. We show one example in Figure 2.35.

Eftersom E är begränsad enligt lemma 2.41, har (a_k) en konvergent del-
följd (a_{n_k}) enligt Bolzano–Weierstraß sats (sats 2.23). Låt A beteckna delföljd-
dens gränsvärde. Vi vill visa att $A \in E$.

traditional

par passes

Figure 2.35

It is possible to disallow line breaks before short math formulas by manually insert-
ing a maximum penalty. Knuth calls them "ties", and it has become standard to use
a tilde to type them, as in Om~A. With our math penalties setup we do not need that
manual tweak in the source. Compare the results in Figure 2.36, where the stricter
math penalties avoids short formulas at the beginning of lines.

Funktionsbegreppet är oerhört viktigt och centralt inom matematiken. Om
A och B är två mängder, så tänker vi ofta på en funktion från A till B som
en regel som till varje element i A tilldelar ett entydigt bestämt element i
B. Om f är en funktion från A till B, så skriver vi $f\colon A \to B$. Det element
i B som funktionen f tilldelar ett element $a \in A$ betecknas $f(a)$. Mängden
A kallas *definitionsmängden* för f och mängden B kallas *målmängden* eller
värdeförrådet till f. Ibland kommer vi att skriva D_f för definitionsmängden
för f.

traditional

Funktionsbegreppet är oerhört viktigt och centralt inom matematiken.
Om A och B är två mängder, så tänker vi ofta på en funktion från A till B
som en regel som till varje element i A tilldelar ett entydigt bestämt element
i B. Om f är en funktion från A till B, så skriver vi $f\colon A \to B$. Det element i B
som funktionen f tilldelar ett element $a \in A$ betecknas $f(a)$. Mängden A
kallas *definitionsmängden* för f och mängden B kallas *målmängden* eller
värdeförrådet till f. Ibland kommer vi att skriva D_f för definitionsmängden
för f.

par passes

Figure 2.36

Line breaks inside formulas can become very ugly, compare the results in Fig-
ure 2.37.

The orphan penalties sometimes help us to prevent just a word or a symbol at the
end of the line. Compare the results in Figure 2.38.

We also show one example of a paragraph where TeX fails with the traditional runs
but succeeds with the par passes, see Figure 2.39.

Lösning. Eftersom $a^{1/k} = 1/(1/a)^{1/k}$ räcker det att visa påståendet för $a > 1$. För $a > 1$ och $k \in \mathbb{N}$ är $a^{1/k} > 1$. Bernoullis[11] olikhet (se övning 2.4) med $x = a^{1/k} - 1$ ger

<div align="center">traditional</div>

Lösning. Eftersom $a^{1/k} = 1/(1/a)^{1/k}$ räcker det att visa påståendet för $a > 1$. För $a > 1$ och $k \in \mathbb{N}$ är $a^{1/k} > 1$. Bernoullis[11] olikhet (se övning 2.4) med $x = a^{1/k} - 1$ ger

<div align="center">par passes</div>

<div align="center">Figure 2.37</div>

Ibland vill man starta summationen på något annat tal än 1. Frågan om konvergens/divergens för serier beror inte på hur de första termerna uppför sig, utan endast hur a_k beter sig för stora värden på k. Vi kommer därför att använda beteckningen $\sum a_k$ för att beteckna serien med termer a_k, där den undre gränsen är något fixt heltal (ofta 0 eller 1) och den övre gränsen är $+\infty$.

<div align="center">traditional</div>

Ibland vill man starta summationen på något annat tal än 1. Frågan om konvergens/divergens för serier beror inte på hur de första termerna uppför sig, utan endast hur a_k beter sig för stora värden på k. Vi kommer därför att använda beteckningen $\sum a_k$ för att beteckna serien med termer a_k, där den undre gränsen är något fixt heltal (ofta 0 eller 1) och den övre gränsen är $+\infty$.

<div align="center">par passes</div>

<div align="center">Figure 2.38</div>

In addition to the previously mentioned settings for the math book, we also use

```
\setupalign
  [hanging,      % protrusion
   depth,        % align stuff at bottom of page
   profile,      % even line spacing if possible
   granular,     % a granular setup (classes and adjacentdemerits)
   lesswidows,   % a strict widow setup
   lessclubs,    % a strict club setup
   lessorphans,  % a strict orphan setup
   lessbroken,   % try to avoid hyphenations over pages
   strictmath]   % strict math penalty setup
```

The `lessbroken` option makes sure that we penalize hyphenations that run from right to left pages more than from left to right pages. This make sense in a book, where one has to turn pages.

Serien $\sum (-1)^k/k$ är konvergent, och eftersom $0 < a_k < 1/k$ så är serien $\sum (-a_k)^k/k$ absolutkonvergent eftersom $\sum 1/k^{k+1}$ är konvergent (jämför till exempel med den konvergenta p-serien $\sum 1/k^2$). Det följer att serien $\sum (-1)^k a_k$ är konvergent. Alltså konvergerar $\sum a_k x^k$ precis då $-1 \le x < 1$.

traditional

Serien $\sum (-1)^k/k$ är konvergent, och eftersom $0 < a_k < 1/k$ så är serien $\sum (-a_k)^k/k$ absolutkonvergent eftersom $\sum 1/k^{k+1}$ är konvergent (jämför till exempel med den konvergenta p-serien $\sum 1/k^2$). Det följer att serien $\sum (-1)^k a_k$ är konvergent. Alltså konvergerar $\sum a_k x^k$ precis då $-1 \le x < 1$.

par passes

Figure 2.39

Let us also mention that these extra par passes do not increase compilation time much. We have kept track of some of the compilation times for the book (290 pages) while working. The par passes were enabled in the September 2023 run (but the setup has been evolving).

```
December  2022 | total runtime: 12.109 seconds
July      2023 | total runtime: 7.997  seconds
September 2023 | total runtime: 8.306  seconds
April     2024 | total runtime: 9.739  seconds
April     2024 | total runtime: 18.439 seconds (with synctex and
                                                          tagging)
September 2024 | total runtime: 9.290  seconds
```

So, it has not slowed down much. This is of course non-scientific, but the runs were done on the same computer. If it was only the par passes that had changed (it is not), one second extra is not a big deal. With synctex and tagging enabled, the compilation time doubles.

2.17 Another test paragraph

In D.E. Knuth's *Digital Typography* he uses a rather math-dense paragraph as a showcase. We display that paragraph below, in a few different text widths, with the traditional run, and with the par passes from the last section, used in the math book.

In Figure 2.40 we note that the traditional paragraph builder is hyphenating and breaking inside one of the formulas. The third subpass manages without both.

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

traditional

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

par passes

Figure 2.40   A test paragraph from Knuth's Digital typography. Here `\hsize` is 300pt.

In Figure 2.41 (`\hsize`=240pt), the traditional parbuilder fails, with an overfull line. Here we need subpass 5. The first line is a bit loose, but overall it looks good.

In Figure 2.42, the text block is quite narrow (180pt). The traditional builder fails again, while subpass 5 succeeds. We get another broken formula and a few hyphenated lines, but given the constraints it is not too bad.

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

<div align="center">traditional</div>

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

<div align="center">par passes</div>

Figure 2.41   A test paragraph from Knuth's Digital typography. Here \hsize is 240pt

2.18 Summary

We have discussed an extension to the traditional Knuth–Plass paragraph builder, implemented in LuaMetaTEX, and available today to ConTEXt users.

The main new feature is the possibility of having an arbitrary number of runs over each paragraph, with independent setups for each run. With a setup where the first two runs are similar to the traditional pretolerance and tolerance runs, most ordinary paragraphs are taken care of by them, leaving only the more difficult paragraphs to be handled by the later runs. This means that the impact on speed is negligible.

We have also introduced a few new penalty and demerit parameters, and made others more configurable, with plural versions and sometimes also with the possibility to keep track of left and right pages.

The next step is to test this on various types of documents and to provide a few standard setups that make sense for the users who don't want to mess with details. There are already a few users who are up and running on their book projects, and they seem to be very satisfied.

We also touched upon how the result of the paragraph builder can influence the page builder. The process of building pages is at a first glance simpler than the building the

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

<center>traditional</center>

Figure 2.42 A test paragraph from Knuth's Digital typography. Here `\hsize` is 180pt

paragraphs, since we merely add content to a vertical list until it is deemed as being full, and then ship out the page. On the other hand one has to handle footnotes, floats, sections, columns, and so on, and that greatly complicates the matter. We intend to study this process in a future project.

3 Twin demerits

This chapter was written for the TugBoat and appeared as preprint in the proceedings of the 2024 ConT_EXt meeting. Many thanks to Karl Berry who, as usual, improved the writing a lot and also gave valuable feedback on the confusing bits of the content. Thanks Karl!

Upgrading math support in ConT_EXt not only concerns rendering formulas but also breaking formulas across lines. For instance, fenced formulas should cross lines while retaining the automatic scaling of fences but at the same time you don't want a single fence at the beginning or end of a line. Longer formulas should preferably break somewhere away from the begin and start. Single atoms should not end up at the end of a line and the same is in fact true for text. The later can be prevented by so-called toddler penalties. And then there are languages where (binary) operators need to be repeated, in a similar way as hyphens, at the start of a broken line. Alternative content (swapping one word for another in order to get a visually better looking paragraph) is also possible but those are more (usable) proofs of concept than features used daily. We are fans of the 'rewrite if needed' approach, but it is of course a nice and fun challenge to solve some typographical problems in a generic way, when possible.

So, looking at the end of a broken line and the beginning of the following becomes second nature when moving forward with development. In order to explore and test all these possibilities, we added ways to trace the process of breaking lines: we love to visualize such things. When playing with this we also looked at the start and end of lines with repeated sequences, for instance avoiding the same words or math variable stacking at the start or end, similar to repeated hyphens. But we had enough on our plate to not fully explore this beyond some experiments.

Around that time we had some email contact with Didier Verna, who at the 2023 tug meeting reported on some experiments he conducted in the ETAP (Experimental typesetting algorithms platform) software that he develops. He followed up on that in 2024 with a preprint of an article reporting on further experiments, especially avoiding similar words at the start and end of successive lines.[4] Of course given the long history of T_EX it is no surprise that the wish to avoid that has been expressed before, but this was the first time we have seen detailed data on the topic. We already knew that extensions to the par builder didn't come at a huge performance hit and Didier, also knowing this, therefore wondered if adding such prevention to the engine was an option.

---

[4] Similarity Problems in Paragraph Justification, An Extension to the Knuth-Plass Algorithm, Didier Verna, EPITA Research Laboratory, Le Kremlin-Bicêtre, France, July 2024 (preprint).

Before we dive into this one should notice that over time many suggestions have been made with regards to where TEX can be improved. Among the reasons why these never made it into the engine are that TEX is frozen, so extensions have to go into successors like ε-TEX, pdfTEX, X∃TEX, LuaTEX, LuaMetaTEX, etc. One complication is in the way TEX is programmed: it uses a linked list of nodes for what eventually becomes a list of lines, a paragraph. This is a forward-linked list so one cannot look back, although in some cases TEX keeps a pointer to the previous node around. But looking back a 'word' demands quite a bit of extra code. In LuaTEX and LuaMetaTEX we have a dual linked list so there we can go back. This means that implementing a feature as discussed here is less hard and also can be prototyped in Lua. On top of that, in LuaMetaTEX we also carry around more information to act upon. Of course it doesn't change the fact that while experiments can show that 'it can be done' doesn't mean that we don't run into complications that have to be dealt with in order to make it usable and not backfire with bad results elsewhere. We will show a few cases that demonstrate that one reason for engines not to support this out of the box is that for a single (extra) feature like this, one likely has to add far more control options. So keep this in mind when reading on: there is always more involved than at first sight. What looks like a home run for LuaMetaTEX is less so for other engines.

As we had been playing a bit with tracing and analysing decisions we have a mechanism in place to plug code into the par builder. We use this for instance to force breaks based on feedback. Discouraging a break at similar words can be done using those same hooks so we decided to take the challenge and just made it a more permanent feature, possibly with the side effect of it being more integrated in the engine.[5] Below are some outcomes that can be seen as a progress report on this feature.

The first thing we did was go back to the already existing callback. Because the builder is rather complex (keep in mind that we have several extensions) there are nine places where the callback can be triggered, mysteriously identified as: initialize, start, list, stop, collect, line, delete, report, and wrap up. Each call to the same callback gets a different set of status parameters that at that particular moment make sense. It is up to Lua code to collect, analyze, feedback and/or use it somehow. This plugin mechanism seems like a lot of overhead but as it is only needed for tracing it goes unnoticed.

When we play with these repeated words we distinguish between what we call left and right edge twins.[6] We look at glyphs as well as discretionaries and ignore font

---

[5] We often keep experimental code around, interfaced not at the user level but via runtime directives, if only because we need it for articles. Supporting a feature as discussed here needs some thinking with respect to integrating in, for instance, the paragraph rendering setups which differs from low-level directives.

[6] So in addition to widows, clubs, toddlers and orphans we now have twins too.

kerns. We need to check the pre, post and replacement parts of discretionary nodes because we must assume that more complex OpenType features might give a more complex discretionary than a single hyphen.[7]

Using a Lua approach is quite flexible and permits nice tracing but, as said, it abuses a callback that, due to the many different invocations, is not the best candidate for this. We could add another callback but that is overkill. Therefore, after testing, part of the Lua code has been turned into a native feature so that we can do both: native twin checking as well as tracing of the break routine (which we need for testing par passes) but also exploring more variants using that callback.

So, the reference implementation is still done in Lua where we then also have twin tracing. In principle that is fast enough; the overhead on a 240 page (1000) Tufte quote test is around .1 seconds. The native C implementation works slightly differently but is derived from the Lua code. In the engine we have some constraints, such as limiting the maximum length of a snippet to 16 characters.

In both cases (Lua and C) the overhead is rather small because we look only at a limited set of breakpoints. In Lua we gain performance by caching, in C by limiting the snippet. We can squeeze out some more performance if needed by immediately comparing the second snippet with the first one. Unlike the Lua variant, the C implementation checks for a so-called glyph option being set.[8] Because it has to fit into how we handle linebreak controlling parameters, we carry new `\lefttwindemerits` and `\righttwindemerits` registers in the paragraph state node and we can also set it in the (optional extra) paragraph passes, so that it can be disabled when we get bad results. This makes a relatively small patch a bit larger due to housekeeping.

With support in the engine (C) as well as in Lua (the callback), we can now come back to some of the observations we made when we discussed this feature during experiments. But first let's stress that adding this feature to the engine makes sense so that users can play with it, but this doesn't mean that it always solves the problem. Also, like other features, one might only benefit in a few places in hundreds of pages of text. One should always visually check the result.

In his article Didier uses a quote (from Grimm Brothers' "Frog King") that in his case has three 'and''s in row (using an eight bit Computer Modern font). Actually there are four 'and''s close together that can team up. Here we use a different setup with the same quote. We have different defaults for e.g. tolerance and spacing in ConTEXt anyway. In figure 3.1, we start with the paragraph as it comes out normally, using 12 point Latin Modern and an `\hsize` of 82mm.

---

[7] In his preprint Didier only mentions glyphs and stops in his experiments at discretionary nodes.

[8] Glyph options control features like kerning, ligature building, protrusion, expansion, at the individual glyph level.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.1

In figure 3.2 we show what we get when we set the demerits to 7500 thereby enabling twin detection. This number is pretty high because demerits are usually large numbers, as in squared penalties. When a paragraph is broken into lines TEX keeps track of reasonable breakpoints. As it goes over the paragraph breakpoints get identified and depending on criteria previous breakpoints get looked at. That means that at any of these points we can check if there are similar words before and/or after a pair. If that is the case one or both extra demerits get added to the current accumulated amount. Normally the current amount is in the thousands so that is why we need relatively high twin values.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.2   Twin demerits parameters set to 7500, engine implementation.

In figure 3.2 we use the engine variant; in the next example we use the Lua implementation, which permits coloring the snippets that we found troublesome. Tracing also happens on the console and that is why we use the callback: if we report the

words that matter, we need a proper Unicode string and in a typeset paragraph we might have (in the case of ConTeXt) private ones that point to ligatures, case variants, stylistic alternates etc.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.3   Lua implementation, with colored snippets.

Notice that we not only detect an 'and' case here but also a hyphenated part of 'forest'. Of course the whole 'forest' could also have shown up as a candidate.

All this depends a lot on the fonts and widths used. In figure 3.4 we use the Pagella font. It demonstrates that one cannot simply assume that when twins get set up the desired effect occurs. Again we set the values to 7500, and in figure 3.5 we get the same results, contrary to the Latin Modern case.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.4   The Pagella font, without twin detection.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.5   The Pagella font, with twin detection, but line breaks are unchanged.

Figure 3.6 uses the Lua variant so that we can show the candidates, red for the right ones; later we'll also see green for the left ones and yellow for both left and right. In all cases, words are colored when they were considered as twins at some point in the paragraph processing, even if those particular line breaks were discarded later. Thus, the colored words might show up anywhere in a paragraph.

At any rate, the reason why it doesn't work out here is that we need to bump the tolerance and also permit emergency stretch. This shows that just enabling a feature doesn't guarantee results. So, figure 3.7 does that: more tolerance and possible emergency stretch. Playing with the widths shows that single point differences can have quite some effect.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.6   Showing candidates in the Pagella example.

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Figure 3.7   With more tolerance
and emergency stretch.

Finally we show a few examples with nonsense text. The red words are the ones that show up at the right, the green ones on the left, but when a word can occur at both ends yellow is used.

more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient more and efficiency and efficient

Figure 3.8

With the demerits set to 5000 (figure 3.8) we still get a few 'efficien' at the left but they are different words. One can argue that we could use some snippet length (say six glyphs) but of course then something else will bother us. In the next variant of the above, we set \parfillskip such that we have a different solution space, combined with an extreme 25000 demerits (figure 3.9). In both examples we use a 426 point width.

more and efficiency and efficient more and efficiency and efficient more
and efficiency and efficient more and efficiency and efficient more and ef-
ficiency and efficient more and efficiency and efficient more and efficiency
and efficient more and efficiency and efficient more and efficiency and effi-
cient more and efficiency and efficient more and efficiency and efficient more
and efficiency and efficient more and efficiency and efficient more and ef-
ficiency and efficient more and efficiency and efficient more and efficiency
and efficient more and efficiency and efficient more and efficiency and ef-
ficient more and efficiency and efficient more and efficiency and efficient

Figure 3.9 With unusual parfillskip and demerit registers at 25000.

Going narrower, as in figure 3.10, brings us words that can be at either end (shown in yellow) and leaves us without solution but that is what we expect. One can conclude that this feature works best with a wider layout and is not that useful in columns, unless one prefers excessive space glue over no twins, but the good news is that TeX is unlikely to favor that.

more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient
more and efficiency and efficient

Figure 3.10 Narrower hsize.

So what can we conclude? First of all that it is indeed possible to get rid of repetition. To what extent this improves a document while not introducing suboptimal para-graphs we leave to the user; Didier makes a case for it. Performancewise, there is no

reason not to enable it. We did some tests with the larger documents that we also use for testing other features (math line breaking, page building) and when there are twins seen sometimes they do indeed get separated.

One of our test documents, the King James bible in two columns using the Unifraktur font, is a good candidate but although we find candidates only in some cases the line break routine was not always influenced by the increased demerits. Examples of two letter words are 'of' and 'is' and of course it being English we find plenty 'the' and 'and' but some still ended up below each other, simply because we have narrow columns. In figure 3.11 (a bitmap screenshot) we see an interesting case but one that happened to render the same without twin detection. Words like 'their' and 'shall' happily team up as twins, no matter how high we set the demerits.

19 Three bowls made after the fashion of almonds in one branch, a knop and a flower; and three bowls made like almonds in another branch, a knop and a flower: so throughout the six branches going out of the candlestick.

Figure 3.11   Example from the bible.

In the pdfTeX project, font expansion was tested on an annotated bible and the combination of text, notes, numbers, references was a real challenge.[9] In the abovementioned King James we don't have those constraints but one can wonder what setup will make the verse in figure 3.12 come out better. We bet that the double twins here are considered less of a problem than excessive spacing or extreme expansion.

8 And thou shalt bring the meat offering that is made of these things unto the LORD: and when it is presented unto the priest, he shall bring it unto the altar.

Figure 3.12   With perhaps too much font expansion.

The good news is that in this 740 page document, there were quite a few catches, like the one in figure 3.13. In some cases we got one line more or less and therefore a different column or page break. Of course this itself then can create a problem, like a widow or club but we set that up with pretty high penalties and combined with vertical expansion and page slack tolerance (both are relatively new features) we still get good output so overall we gain in quality!

5 And every plant of the field before it was in the earth, and every herb of the field before it grew: for the LORD God had not caused it to rain upon the earth, and there was not a man to till the ground.

Figure 3.13   Overlaying results; the first line ends with a doubled 'the'.

---

9 Hàn Thế Thành's thesis reported on that.

We tested two other documents that show some interesting (challenging) aspects. In Mikael's university course we compared the versions with and without twin control. The tracer identified 25 situations where demerits could be bumped. We noticed that 'att', 'och', 'så', 'vi' and other short words were popular candidates but after turning on tracing we saw that many were left and were surprised to see quite a few longer words, with of course in a math book quite some 'komplexa' and 'negativa' showing up at the edge.

Keep in mind that we only look at a subset of the possible breakpoints. Of these 25 only 5 were applied, so for the other 20 the solution space was not adequate. For the 5 cases the solution resulted in somewhat narrower lines so we wondered if additional par passes made (hz) expansion kick in but it didn't so in the end we're okay. Of the 20 remaining cases 10 had long words, some with hyphenation so actually we had more cases. An interesting side effect of tracing (by color) is that we noticed that the long words also had successive words and that rewriting the paragraph made sense.

element (a, b) där $a \in A$ och $b \in B$. Exempelvis är $\mathbb{N} \times \mathbb{N}$ mängden av alla par av naturliga tal, såsom $(1, 1)$, $(2, 3)$ och $(101, 23)$.

Ett mycket viktigt och centralt begrepp i matematiken är begreppet *funktion*. Om A och B är två mängder, så tänker vi ofta på en funktion från A till B som en regel som till varje element i A tilldelar ett entydigt bestämt element i B. Om f är en funktion från A till B, så skriver vi $f : A \to B$. Det element i B som funktionen f tilldelar ett element $a \in A$ betecknas $f(a)$. Mängden A kal-

Figure 3.14   Math example; the 'alla' on the first line is repeated at the bottom of the previous page.

In a math document sometimes it's unavoidable. In figure 3.14 we see a few trouble-makers and 'alla' is actually not resolved. The figure shows the top of a page and at the bottom of the previous page there's also 'alla'. We don't even want to ponder how to bring page breaks into this model. One can also wonder what is more troublesome: edge cases or middle cases.

Also worth noticing is that when twins end up in the middle they tend to stack even when the par builders in the end didn't consider the end-of-line case anyway. A bad example had three separate slightly offset but still stacked long words, shown in figure 3.15. And, once the author saw this, he made a note to "fix it by rephrasing".

Funktionen $x \mapsto \tan x$, kvoten av $\sin x$ och $\cos x$ är väldefinierad så länge som $\cos x \neq 0$. Det är klart att $\tan 0 = 0$ och att $\tan$ är kontinuerlig och strängt växande på $(0, \pi/2)$ (strängt växande följer av att sin är positiv och strängt växande och cos är positiv och strängt avtagande på intervallet). Då sin är

Figure 3.15   Worse math example.

The 'solved' cases were mostly short words but so were unsolved ones; see figure 3.16. The constraints that math put on breaking the lines win over any twin constraints we add. We also were confirmed in our decision to take discretionaries into account.

Om $0 < a < 1$ så är $1/a > 1$ och $a^x = 1/(1/a)^x$. Alltså följer det från det vi just gjort att $(1/a)^x \to +\infty$ då $x \to +\infty$, dvs. givet A existerar ω så att $(1/a)^x > A$ om $x > \omega$. Givet ett godtyckligt $\varepsilon > 0$ finns det alltså ω så att $|1/(1/a)^x - 0| = 1/(1/a)^x < \varepsilon$ om $x > \omega$. Alltså gäller det att $a^x = 1/(1/a)^x \to 0$ då $x \to +\infty$. ∎

Figure 3.16   Math example.

We also tested a document that Mikael typeset from Gutenberg sources for a book club, Henry James' *The Turn of the Screw*. Here we again noticed quite a few duplicates but also quite a few eventually separated twins, as in figure 3.17.

—superficially at least—by a visible wound to his head; such a wound as might have been produced—and as, on the final evidence, had been —by a fatal slip, in the dark and after leaving the public house, on the steepish icy slope, a wrong path altogether, at the bottom of which he lay. The icy slope, the turn mistaken at night and in liquor, accounted for much—practically, in the end and after the inquest and boundless chatter, for everything; but there had been matters in his life—strange passages and perils, secret disorders, vices more than suspected—that would have accounted for a good deal more.

Figure 3.17   From "The Turn of the Screw".

In figure 3.18 we wondered if the twin handler had kicked in which indeed was the case. But we also noticed that without this mechanism being enabled, the same midline stacking occurred. However, in both cases, without coloring they can easily go unnoticed; just try to locate them in figure 3.19. (See figure ?? for the results.)

This document also demonstrated that words close together tend to register as siblings, and when Mikael showed one of his children what we were looking at, she noticed disturbing repetitions which we hadn't noticed before.[10] But adding more tricky mechanisms will only make the solution space smaller so we will not reveal every annoyance. We did once consider \siblingpenalty but already forgot what for, but we hereby reserve that name.

There is plenty left to explore. It is not uncommon in the T_EX community to hear users (and developers) express the wish for a feature, offer a few examples of why it's needed, and then fall silent. Time and money can be arguments used to not spend

---

[10] From figure 3.20 you can deduce what words were involved. In that example there are many possible twins, so we set twinslimit to 3, a feature added for this purpose to the Lua version.

settled: there was a queer relief, at all events—I mean for myself in especial—in the renouncement of one pretension. If so much had sprung to the surface, I scarce put it too strongly in saying that what had perhaps sprung highest was the absurdity of our prolonging the fiction that I had anything more to teach him. It sufficiently stuck out that, by tacit little tricks in which even more than myself he carried out the care for my dignity, I had had to appeal to him to let me off straining to meet him on the ground of his true capacity. He had at any rate his freedom now; I was never to touch it again; as I had amply shown, moreover, when, on his joining me in the schoolroom the previous night, I had uttered, on the subject of the interval just concluded, neither challenge nor hint. I had too much, from this moment, my other ideas. Yet when he at last arrived, the difficulty of applying them, the accumulations of my problem, were brought straight home to me by the beautiful little presence on which what had occurred had as yet, for the eye, dropped neither stain nor shadow.

Figure 3.18   Another text from "The Turn of the Screw".

was a queer relief, at all events—I mean for myself in especial—in the renouncement of one pretension. If so much had sprung to the surface, I scarce put it too strongly in saying that what had perhaps sprung highest was the absurdity of our prolonging the fiction that I had anything more to teach him. It sufficiently stuck out that, by tacit little tricks in which even more than myself he carried out the care for my dignity, I had had to appeal to him to let me off straining to meet him on the ground of his true capacity. He had at any rate his freedom now; I was never to touch it again; as I had amply shown, moreover, when, on his joining me in the schoolroom the previous night, I had uttered, on the subject of the interval just concluded, neither challenge nor hint. I had too much, from this moment, my other ideas. Yet when he at last arrived, the difficulty of applying them, the accumulations of my problem, were brought straight home to me by the beautiful little presence on which what had occurred had as yet, for the eye, dropped neither stain nor shadow.

Figure 3.19

time on actually implementing something and the possibility keeps floating around. One can play science and stop an experiment with the usual "suggestions for further research" and move on. It's therefore nice to see some real research on the topic as with Didier's using a prototype. However, because typesetting is pretty much about esthetics and boundary conditions we have to face reality and that's what we hit when testing. An example is the following case:

```
....  \im {x+1}.
....  \im {x+2}.
```

saying that instead of growing used to them—and it's a marvel for a governess: I call the sisterhood to witness!—I made constant fresh discoveries. There was one direction, assuredly, in which these discoveries stopped: deep obscurity continued to cover the region of the boy's conduct at school. It had been promptly given me, I have noted, to face that mystery without a pang. Perhaps even it would be nearer the truth to say that—without a word—he himself had cleared it up. He had made the whole charge absurd. My conclusion bloomed there with the real rose flush of his innocence: he was only too fine and fair for the little horrid, unclean school-world, and he had paid a price for it. I reflected acutely that the sense of such differences, such superiorities of quality, always, on the part of the majority—which could include even stupid, sordid headmasters—turn infallibly to the vindictive.

Figure 3.20 Example of many twins, with twinslimit=3.

In the paragraph stream we get math formulas followed by a period. However, what we really get after the '1' and '2' is a math end node, a penalty, and a (likely zero) glue or kern (depending on what we configured). This means that the period is seen as a snippet and so we get a twin here, and bumping demerits then interferes with our rather advanced math spacing and penalty model. This made us be more strict in what makes for a possible sibling: we expect glue and glyph after and/or glyph and glue before. Maybe we should be even more restrictive and look at character properties which makes us end up in Lua.

Another challenge is shown in figure ??, where we have twins that are followed by punctuation. So how do we tackle that? At the Lua end we have access to the font properties so there we can act on the original Unicode character being punctuation, in which case we can ignore it. At the TeX end we need to figure that out differently. We could look at the `\sfcode` but that's rather unreliable. We could have a callback that gives the required property information, but do we really want an extra callback? In the example the third paragraph is done by our Lua implementation. The second one comes from the engine where we use an experimental character control feature that we set up for this case.[11] The verdict is still open if we add this feature, also because for it to be useful yet another field in the glyph node would be required.

So, as we move on and test more, additional constraints can occur. It is easy to come up with various "TeX should do this or that", or even "I looked into it and it can be done", and then end up with "Sorry, not now." It does take time and effort indeed but it also brings one into unknown territory. So, we do show that it can be done but we will never claim that what we do is perfect and we definitely do not enable it by

---

[11] Think of cccode"2E = "0001 (period) and cccode"2C = "0001 (comma) that sets the ‚ignore twin' bit, where cccode is the ``character control'' primitive.

test even more test more, and test
more, and test even more test more,
more test more, and test even more
test even more

test even more test more, and test
more, and test even more test more,
re test more, and test even more test
even more

test even more test more, and test
more, and test even more test more,
re test more, and test even more test
even more

Figure 3.21   Twins with punctuation. First paragraph has default processing; second with an experimental engine feature, third with Lua.

default. It will take some time and likely input from ConTEXt users to fine-tune this, assuming it gets used. It can currently be enabled by setting one of the align options:

```
\setupalign[notwins] % for the brave:
                              [notwins,notoddlers,noorphans]
```

Let's end with some statistics. In this document we enable multiple par passes, but the number of times that these are needed is small. The extra overhead can often be neglected anyway. Here's how many first, second and emergency passes we have and how often additional sub-passes were needed to fit the criteria. In the King James we bumped the demerits by 7500 for 665 left twins, 772 right twins and 113 of these end up left and right.

context	first	second	emergency	sub-pass
page	35989	4733 (13%)	0 (0%)	282 (1%)
vbox	2942	734 (25%)	0 (0%)	0 (0%)

The document has 246,470 words, of which 112,329 get hyphenated in 35,750 checked node lists. A run without twin detection takes 14.50 seconds, with engine twin detection that gets raised to 14.75 seconds. Because here we have only text and many small paragraphs the Lua variant performs relatively slowly: 15.35 seconds. Tracing, marking words with color and reporting to the console adds .15 seconds to that. This document is not the fastest to process: we use columns, a rather demanding font, selective expansion (sub-pass driven), and the sources are xml which gets interpreted and remapped on the fly.

Thanks to Didier for inviting us to prove that it can be added to the engine with little effort and providing some stimulating statistics. Let's end with some more because it can't be that there is no performance hit when we enable this feature, right? So let's check out three scenarios:

1. The `\glyphoptions` variable has the 'checktwin' bit set but both twin demerits parameters are zero, so we never enter the check.

2. The `\glyphoptions` variable has the 'checktwin' bit set and both twin demerits parameters are 7500. We enter the check and per-glyph options permit it.

3. The `\glyphoptions` variable has the 'checktwin' bit unset but both twin demerits parameters are 7500. We enter the check but per-glyph options prevent it from succeeding.

In ConTEXt we set the demerits and use the options bit to control it, so we always have the check but can quit after some initial tests (case 2 and 3). The numbers below are for ten runs of 15000 times each of the well-known Tufte quote, for each of the three cases:

```
\setbox\scratchbox\vbox{\samplefile{tufte}}}
```

1	17.860	18.478	19.026	18.824	18.736	18.665	18.623	19.002	18.101	18.905	18
2	18.672	19.181	18.150	18.960	18.414	19.120	18.246	18.945	19.050	18.744	18
3	18.979	18.597	18.747	18.837	18.660	18.846	18.513	18.457	18.448	18.414	18

The results are in table table:stats. These numbers include font processing time as well as some other ConTEXt specific callback overhead processing time but we want to test with ligatures and discretionaries so this is required. When we use `\vpack` all times are the same.

But, this is for 15000 nine-line paragraphs using the Tufte quote and that is a tough one: many short words, ligatures, four hyphenated lines in the standard layout. If we output the result, we get a 3335 page document and a runtime of about 37.5 seconds (on my 2018 laptop).

1	no check at all	**18.622**	37.270	37.433	37.425	**37.376**
2	check and honored	**18.748**	37.651	37.261	37.690	**37.534**
3	check but ignored	**18.650**	37.032	37.565	37.967	**37.521**

So, in the end, assuming that we have the third variant as default (which is the most practical in ConTEXt) users will see a small performance hit due to this new feature but on a regular run, which in practice does way more than just outputting text only, no one will notice it. So, our and Didier's conclusion that we have no performance hit (something that is always considered when making a possible extension to a core component) holds.

you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you you (as a daddy) are never too old to learn from young kids are you

Figure 3.22

4 Namespaces

Occasionally on TeX related mailing lists, meetings, articles or forums performance comes up. It makes no sense for me to go into the specific (assumed) bottlenecks mentioned but as in ConTeXt we do keep an eye on performance every now and then I also spend words on it, so here are some.

The nature of the (multilingual) user interface of ConTeXt there is extensive use of the `\csname` and related primitives. For instance, if we have the namespace `999>` and a keyword `testkeyword`, we can have a specific property set with:

```
\expandafter\def\csname 999>testkeyword\endcsname{}
```

We can then test if a macro with the inaccessible name '999>testkeyword' exists and has been set with a test command available in all engines that carry ε-TeX extensions:

```
\ifcsname 999>testkeyword\endcsname
    % whatever
\fi
```

In order to test this, the list of tokens starting at `9` and ending at `d` has to be converted into a (C) string that is used for a hash lookup. One can expect this to be a costly operation. In a 300 page book with many thousands of formulas this easily runs into the millions. Testing this five times on one million such tests gives:

```
0.303 0.293 0.283 0.301 0.298
```

for LuaMetaTeX and

```
0.276 0.287 0.287 0.274 0.274
```

for LuaTeX. I deliberately show five numbers because one has to keep some system load into account. When I'm interested in performance I only care about trends because no run ever gets the whole machine for its job. That said, where does the noticeable difference between these engines come from? It can partly be explained by LuaMetaTeX having more primitives and therefore a bit more overhead (more scattered code in memory and cpu cache). But as the basic code that kicks in here is not that much different I figured that it might be the hash lookup and, because indeed we had a follow up lookup in the hash (two steps), by using a larger hash table we could limit that to a direct hit.

```
0.288 0.281 0.280 0.288 0.277
```

So we ended up with similar measurements for these engines. Before we carry on, let's ask ourselves if these numbers worry us. Say that this book takes 12 seconds to process, does it matter much if we half this overhead? Probably not, but in the following, we need to keep in mind that much can interfere. A simple million times test is likely very cpu cache friendly. There are however other factors in play: convenience coding, abstraction, less cluttered tracing, more detailed feedback from the engine, less code and memory usage, the size of the format file. Trying to get lower numbers is also kind of fun.

Back to the user interface, we now introduce some abstraction (the `test` in the names avoids clashes with existing definitions):

```
\def\??testfoo     {999>}
\def\c!testkeyword{keyword}

\ifcsname\??testfoo\c!testkeyword\endcsname
    % whatever
\fi
```

Again LuaMetaTEX is a little slower but it is kind of noise:

```
0.243 0.243 0.247 0.241 0.249 luatex
0.251 0.250 0.250 0.249 0.249 luametatex
```

But how about the following timings for LuaMetaTEX:

```
0.136 0.143 0.139 0.139 0.140
0.132 0.132 0.133 0.129 0.130
```

In the first case we defined the namespace and keyword as follows:

```
\cdef\??testfoo     {999>}
\cdef\c!testkeyword{keyword}
```

A `\cdef`'s macro is basically an `\edef`. This definition is scanned as token list and therefore we know the macro has no arguments. It operates as any macro but in a `\csname` related command it is just passes as-is and only expanded when we need to do a lookup. When that happens we don't need to go through a token list (copy) but directly can go to string characters.

The second measurement shows a little improvement and is the outcome from an experiment with build in namespaces. Think of this:

```
\namespaceifcsnamedef\iftestfoocsname 999
```

```
\iftestfoocsname\c!testkeyword\endcsname
    % whatever
\fi
```

That variant is faster but we're talking .05 second on 2.5 million calls in the book because we already use `\cdef`. Even more important is to notice that most documents have only tens of thousands such calls. And 0.15 seconds csname "test and call" on the whole run is not that bad. So, if we go beyond `\cdef` usage we don't need the efficiency argument but the other ones. So, after a few days of playing with this I rejected this solution. First of all the source didn't become more readable. We also had many more commands because there were for instance:

```
\namespacecsnamedef      \csnamefoo       999
\namespacedefcsnamedef   \defcsnamefoo    999
\namespaceifcsnamedef    \ifcsnamefoo     999
\namespacebegincsnamedef\begincsnamefoo 999
```

We also had a callback for reporting associated names when tracing. Of course there can be use cases where we have tens of millions of `\csname` calls but I still need to find them. But don't expect miracles now that we're in these low numbers. Integrating all this is also not that trivial because TeX has two separated code paths for expandable commands and ones more related to housekeeping and typesetting (the mail loop). This means that one has to intercept expansion of encoded namespaces and that gives a bit of a mess, especially because we also need to handle nested csnames.

As an aside I also played a bit with 'compiling' regular csname commands followed by a namespace into one token but that was even more messier.[12] So in the end I removed all that experimental namespace code and happily accept the fact that there's nothing to gain, but it was a fun experiment.

As a side effect of this experiment I decided to enable a primitive that had been commented. When it was tested years ago there was no real gain but I realized that it could be implemented a bit more efficient in specific scenarios. Think of this:

```
\csname\ifcsname999>foobar:width\endcsname999>foo:width\fi\endcsname
```

when abstracted becomes:

```
\csname\ifcsname\??testme foobar:\c!width\endcsname\??testme
                                 foo:\c!width\fi\endcsname
```

---

[12] Occasionally I consider some compilation of tokens lists into more efficient ones but so far I could resist.

In both cases the same list of tokens (`\??testme foobar:\c!width`) has to be converted into a byte string, which we can avoid by:

```
\csname\ifcsname\??testme
                foobar:\c!width\endcsname\csnamestring\fi\endcsname
```

when we have a hit. After all, the found macro has a known name that has been registered as a string. This variant runs over 10 percent faster, which of course can be neglected, especially if we don't call it millions of times; the book has 400.000 calls to `\csnamestring`. But as with many optimizations: gaining 20 times 0.1 seconds on different subsystems eventually adds up to 20 % on a 10 seconds run for that 300 page, math extensive, book.

When looking at timings one always need to keep in mind that a simple test (in a loop) is very easy on the cpu cache while in a real document there can be more cache misses simply because the cache is limited in size. That is why in practice we often see a bit more positive impact than shown here. In the case of the `\csnamestring` we not only gain a bit on parameter handling but also on some font related operations, but again the gain depends on how many (more complex) font switches happen, which is more likely in for instance manuals.

5 Bonus features

In ε-TeX the plural `\widowpenalties` and friends were introduced. These use, like `\parshape` a node type that varies in size. In LuaMetaTeX we implement the variable part differently which gives more efficient (and recoverable) memory usage. This is needed because we have more such structures, like `\parpasses` that can become pretty large. The basic approach is:

```
\somecommand number entries
```

where the number of entries is multiplied by a constant depending on `\somecommand`. A `\parshape` takes twice the number, and `\widowpenalties` one or two times the number, depending on a passed option indicating if we differentiate between left and right pages. The `\parpasses` primitive takes dozens of named entries separated by a `next` key and ends with a `\relax`.

```
\somecommand number options bitset entries
```

The bitset after the `options` keyword depends on the command. The fact that we have a somewhat generic structure makes that we can also provide a mechanism for storing 'arrays' of integers, dimensions, posits (and maybe some day token lists). Adding this was a cheap bonus feature that needed little extra code. We implement this using the `\specificationdef` command that takes the form:

```
\specificationdef \somename \widowpenalties ....
```

with of course support for structures other than these penalties. An array is defined with

```
\specificationdef \foo \dimen 3 1pt 2pt 3pt
\specificationdef \oof \count 3 1 2 3
\specificationdef \ofo \float 3 1.1 2.2 3.3
%specificationdef \fof \toks  3 {a} {b} {c} % some day
```

You can access the fields like this:

```
(\the\foo  1) (\the\foo  2) (\the\foo  3)
(\the\ofo -1) (\the\ofo -2) (\the\ofo -3)
(\the\oof  1) (\the\oof -2) (\the\oof  3)
```

The negative index starts at the end and a zero index returns the number of entries and out of range values are zero. An array with two entries per row is defined with an option:

```
\specificationdef \foo \dimen 3 options 4
  1pt 1pt
  3pt 2pt
  5pt 3pt
```

This time we use an index and subindex.

```
(\the\foo 3 1, \the\foo 3 2) : (5pt,3pt)
```

If you want an integer and dimension (or float) you can do this, where the four triggers double entries and 32 tells that the first of each pair is an integer:

```
\specificationdef \oof \dimen 3
    options \numexpr 4 + 32 \relax
    5 2pt
    9 3pt
    2 1pt
```

Although ConTEXt has all kind of data structures like this using Lua, the advantage is that when TEX itself manages this grouping works more naturally. Also, these ways of storing and accessing data is extremely runtime efficient. To what extend if will be used in ConTEXt is to be seen, but it can come in handy when we experiment with paragraph and page builder enhancements in Lua that we want to drive from the TEX end. Given:

```
\specificationdef \foo \dimen 3 options 4
  1pt 1pt
  3pt 2pt
  5pt 3pt
```

the Lua call `tex.getspecification("foo")` gives a table like:

```
{
    {  65536,  65536 },
    { 196608, 131072 },
    { 327680, 196608 },
}
```

So we can for instance consider this to be a table of coordinates defined at the TEX end that can be processed at the Lua end.

6 What if . . .

6.1 Introduction

We don't remove features present in T_EX in LuaMetaT_EX, although there are some exceptions in the sense that we delegate some tasks to Lua. Of course we dropped some ε-T_EX and pdfT_EX extensions and kept very little of what Aleph (Omega) added but apart from relaxing the `\long` and `\outer` prefixes we are good. Of course some primitives, like `\ifcsname` behave better and we handle `\par` in math but that's not really influencing the results. As we go forward it can become tempting to replace some functionality that is sort of redundant or will never be used but it should not have consequences for existing user code. Below I will collect some ideas that fit these criteria.

6.2 Penalty lists

The ε-T_EX extensions introduced a multiple penalty approach, like `\widowpenal-ties` and `\clubpenalties`. This have a few side effects. First of all they take a variable amount of values so they need a variable size data structure and that happens to be nodes. Normally there are not that many definitions so the impact that this has on node memory is limited but if you have thousands of different sized lists it might go bad because they don't get reclaimed and/or reused. This is why in LuaMetaT_EX we have a basic so called specification node with a dynamically allocated list. Of course this has some impact on dumping and undumping because we need to handle these nodes in a special way but it can be done reliable. We also have `\specification-def` to predefine the various plurals, par passes and fitness demerits, so that we can conveniently switch between states.

Another side effect is that setting the `\widowpenalties` masquerades the normal `\widowpenalty` and resetting the plural it makes the singular active gain. So we have two mechanisms (the plural and singular) and one really needs to manage both well in order not to confuse users. For instance, when you want no widow penalties at all, you need to disable both.

Related to this is that when a paragraph starts a node is added, it stores the current state and therefore contains various single and plural penalty states that the post line break routine has to check and apply check: first for the plural and when that one is unset, for the singular. It takes memory, time and code.

When we were setting up the new par passes and extended alignment options we also decided to provide keys (options) for controlling the penalties. I then realized that we can actually redefine `\widowpenalty` and alike like this:

```
\permanent\protected\untraced\def\widowpenalty{\widowpenalties\plusone}
```

However we don't want them to be repeated, which is what the plural does when there is no final 'reset' value.

```
\widowpenalties\plustwo 2500 0
```

The way to deal with this is to use an option:

```
\permanent\protected\untraced\def\widowpenalty
  {\widowpenalties \plusone options
                                   \finalspecificationoptioncode}
```

Which of course involved more parsing, so we made that equivalent to:

```
\permanent\protected\untraced\def\widowpenalty{\widowpenalties\minusone}
```

In a similar way we can also define `\clubpenalty`, `\displaywidowpenalty`, which we ignore in ConTEXt because we do display math differently), `\brokenpenalty`, `\orphanpenalty` and `\interlinepenalty`. When doing so we also set the `\untraced` flag so that when we ask the meaning or enable some tracing they are presented to the user as if they ae primitive. The `\permanent` flag will protect them against redefinition when overload protection is enabled. The `\protected` flag makes sure that we don't expand it in for instance an `\edef` situation.

```
\meaningfull\widowpenalty
```

Compare no prefixed:

```
macro:\widowpenalties \minusone
```

with only `\permanent`:

```
permanent macro:\widowpenalties \minusone
```

and with both `\permanent` and `\untraced`:

```
\widowpenalty
```

Actually, the reason why this works out well is because in the context of am integer value, like `\the` the plural already returned the value requested by the integer following the command, so effectively this already did the job:

```
\the\widowpenaltie\plusone
```

73 What if . . .

Removing the primitives and replacing them this way has the advantage of removing related code which simplifies the post line break routine a bit. Making the par nodes smaller also is nice.

Om an average run a user won't notice the difference because these penalties are not consulted that often during a run: basically once for every line. So, with the standard Tufte test renders in 7 lines, 7000 lines using these three scenarios:

```
\normalwidowpenalty   0 \normalwidowpenalties 0
\normalwidowpenalty 500 \normalwidowpenalties 0
\normalwidowpenalty   0 \normalwidowpenalties 3 500 0 0
```

We get the following results for ten times three times 1000 paragraphs:

```
(3.170,3.208,3.109) (3.161,3.310,3.155) (3.120,3.166,3.142)
                                        (3.116,3.134,3.154)
(3.119,3.193,3.208) (3.063,3.060,3.058) (3.166,3.182,3.142)
                                        (3.158,3.054,3.063)
(3.111,3.198,3.070) (3.195,3.242,3.224)
```

I didn't even bother to turn of the music running in the background because that process, although it slows down the runs, averages well. with such tests it are the patterns that matters, the user experience.

So, currently (September 2024) we just redefine these primitives in ConTeXt but at some point we can either alias them already in the engine or just expect the format file to define them as part of the LuaMetaTeX 'primitives' initialization.

6.3 Math italics

This can be completely dropped from the engine. But because we want to demonstrate differences between tweaked and untweaked fonts, we keep it for now. Maybe some day it will be dropped but then we need to fake the examples in older articles when we rerun them or we need to make the examples into images. Sometimes the simple fact of documenting behavior has that side effect. But it would simplify the code a lot!

6.4 Keywords

Occasionally I wonder how much time is wasted on verbose keywords but because this feature has already been very much optimized, there is little to gain. So, for instance replacing the popular:

```
\srule height\strutht depth\strutdp\relax
```

by

```
\srule ht\strutht dp\strutht\relax
```

or even

```
\srule hd\strutht\strutht\relax
```

will save little, also because this happens in already tokenized macros. So while 50.000 times `\normalsrule\relax` takes 0.004 seconds, the two argument one needs 0.001 seconds (passing just one argument indeed takes 0.007 seconds); this number is a reasonable guess for an average complex 250 page document. So, for now I see no reason to go forward with this also because distinguishing between what follow the h, d or w also adds time.

However, because a strut rule in most cases takes both height and depth, so adding a `pair` keyword can safe some noise on tracing, so I tested that:

```
\srule pair \strutht\strutht\relax
```

And although we only save 0.002 seconds on the 50.000 calls I decided to keep that experiment, if only because we have other noise reduction measures elsewhere too. At the same time I decided to default such rules to zero width; for the record, the running height and depth dimensions are used as signals in (math) char struts so these two need to be set explicitly when values are needed.

7 Expressions

Examples of quantities are internal and register dimensions, counters (also referred to as integers or numbers), attributes, glue, and floats. Most commonly used are dimensions and numbers. Assignments happen like this:

```
\dimen0        10pt % an indexed register
\count20      = 10   % the = is optional
\hsize         15cm % an internal quantity
\scratchdimen  10pt % a \dimendef'd indexed register
```

The scanners involved are also used for getting quantities that are arguments to primitives.

The ε-TEX extension came with `\dimexpr`, `\numexpr` and `\glueexpr` which support simple expressions where the order of what is permitted, so this is ok:

```
x\hskip \dimexpr 10pt * 10 \relax x\par
```

But this isn't:

```
x\hskip \dimexpr 10 * 10pt \relax x\par
```

There are a few pitfalls. For instance, this works ok:

```
x\hskip \dimexpr (10 * 10pt) \relax x\par
```

but this doesn't:

```
x\hskip \dimexpr (10 * 10pt) \relax x\par
```

and the reason is that this feature is either looking for a number or an operator and (is kind of an operator here. In order to make that work the integer scanner has to backtrack when it sees a parenthesis but as this scanner is shares in other situations that actually is an error. Alternatively there could be more look ahead which complicates the code and brings a (nowadays small) performance penalty. Keep in mind that expressions were not part of original TEX, the the simple expressions used what was already there. It does the job but one has to be aware of the somewhat weird parsing rules.

However, LuaMetaTEX has additional scanners that accept the same operators:

```
x\hskip \dimexpression 10 * 10pt \relax x\par
```

They have more operators and also handle boolean expression that can be used in tests. Although these scanners likely can be improved on the average they already perform better than the (also optimized) ε-T$_{\text{E}}$X siblings.

One problem with expressions is that they keep looking ahead until they run into something that doesn't make sense, like `\relax`, or other tokens that are not part of an expression. A common way to end a `\dimexpr` us to use `\relax` but take this:

```
x\hskip \dimexpr 10pt * #1 \relax x\par
```

What if `#1` itself is an expression that doesn't end with a `\relax`? In that case scanning continues after one us seen. It is for that reason that `\dimexpression` also handles braced expressions. When adding some flexibility to the ConT$_{\text{E}}$Xt user interface, by moving (optional) explicit expressions in values to keys that set quantities to the lower level handlers, it started making sense to replace:

```
\def\foo#1{\scratchdimen\relax} \foo{\dimexpr10pt + 1ex\relax}
```

and actually deep down we often already had:

```
\def\foo#1{\scratchdimen\dimexpr#1\relax} \foo{\dimexpr10pt +
                                                1ex\relax}
```

which gives double expression scanning overhead. If we change to the other scanner we get:

```
\def\foo#1{\scratchdimen\dimexpression#1\relax} \foo{10pt + 1ex}
```

or, given that we can use braces:

```
\def\foo#1{\scratchdimen\dimexpression{#1}} \foo{10pt + 1ex}
```

But if we do this frequently do we really need the explicit expression primitive. A few line patch made this possible:

```
\def\foo#1{\scratchdimen{#1}} \foo{10pt + 1ex}
```

and even this:

```
\def\foo#1{\scratchdimen{#1}} \foo{{10pt  +  1ex}}
\def\foo#1{\scratchdimen{#1}} \foo{{10pt} + {1ex}}
```

How dangerous is this from the perspective of compatibility? We'll see but as users normally get an error when doing this in an other engine, it's unlikely that they expect an error here, unless they enjoy triggering errors. There is one side effect worth mentioning:

```
\the\dimexpr {1 + 4}
```

actually works, as does

```
\the\dimexpr -{1 + 4}
```

This is so because numbers scan for expressions, so

```
\number -{1 + 4}
```

is indeed valid. Of course the `\dimexpr` will keep scanning till it sees a `\relax` or something it considers no operator or number.

Once the decision was made to switch to the new expression parser at the TEX end quite some files were affected which is a delicate operation. In the process glue expressions also had to accept the braced variant, something that had been postponed. Also, because we don't have a 'new' parse for glue, we need nested braced scanning there too.

There are more features in LuaMetaTEX that are yet sparsely used but eventually will decorate the code base. An intended side effect is less clutter and less tracing noise but that's more a concern for developers than users.

So what is supported? The usual operators `+`, `-`, `*` and `/` are of course handles. We also interpret `:` (or `div`) and `;` (or `mod`). Bitwise operators are `|` (or `bor` or `v`), `&` (or `band`, `^` (or `bxor`, `!` (or `bnot`, `bset` and `bunset`. The conditional `cand` and `cor` result in a value instead of zero if the condition succeeds. We can shift with `<<` and `>>` and compare with `<`, `<=`, `=` or `==`, `<>` or `!=`, `and` (or `&&`) as well as `or` (or `||`). Negation happens with `not`, `!` and `~` for bitwise operations. There are two somewhat odd infix operators: `nmp` (minus plus) and `npm` (plus minus) that result in a negative or positive value and can be used to get the (complement of an) absolute value.

We can have integers, floats and dimensions and use parentheses for sub-expressions. The `\dimexpression` `\numexpression` don't prioritize because they are variants of `\dimexpr` and `\numexpr`. The `\dimexperimental` and `\numexperimental` cousins actually do prioritize and internally create an RPL stack. In the future we might switch to that alternative when scanning integers and dimensions.

There are some aspects that you need to keep in mind. Compare these two. In the first case we stay within the dimension space which means that the `1pt` and `2pt` are dimension results and therefore get serialized as such, including the zero:

```
\scratchdimenone 100pt \scratchdimentwo 200pt \todimension {
    (\scratchdimenone > \scratchdimentwo) cand 1pt cor 2pt
```

```
}
```

So we get: 2.0pt. In the next case we get what we explicitly asked for: the three token
sequences:

```
\ifdim\scratchdimenone>\scratchdimentwo 1pt\else 2pt\fi
```

So expect this: 2pt. In practice this doesn't matter much and to be honest, even
in ConTEXt using complex expressions is not happening that often, but that might
change over time. By the way, both sides of the 'and' and friends are evaluated, con-
trary to what some programming languages do. On a 2018 laptop, one million itera-
tions of the following give the runtime in seconds shown after the comment. In the
second case we use the ∧ (U+2227) and ∨ (U+2228) symbols which saves some pars-
ing but of course one will probably never do such tests so consider it a bit of 'showing
off'.

```
\scratchdimenone 100pt
\scratchdimentwo 200pt

\scratchdimen % 0.440
    {(\scratchdimenone>\scratchdimentwo) cand 1pt cor 2pt}
\scratchdimen % 0.411
    {(\scratchdimenone>\scratchdimentwo)  1pt  2pt}
\scratchdimen % 0.210
    \ifdim\scratchdimenone>\scratchdimentwo 1pt\else 2pt\fi
\scratchdimen  % 0.209
    \ifdim\scratchdimenone>\scratchdimentwo 1\else 2\fi pt
```

Here are some equivalent operations:

```
\the\dimexperimental{(2   + 1  ) * \lineheight}
\the\dimexperimental{(2.0 + 1.0) * \lineheight}
\the\dimexperimental{(2.1 + 0.9) * \lineheight}
\the\dimexperimental{(1   + 2.0) * \lineheight}
\the\dimexperimental{(2.0 + 1  ) * \lineheight}
\the\dimexperimental{(3        ) * \lineheight}
\the\dimexperimental{(3.0      ) * \lineheight}
```

We get: 51.98428pt 51.98428pt 51.98428pt 51.98428pt 51.98428pt 51.98428pt
51.98428pt , all the same of course. Instead of \lineheight in ConTEXt we can also
use lh because that is one of the units that we define.

There is undoubtely more to say here but that is for the low level manuals to deal
with. Here we just discuss it as some LuaMetaTEX enhancement.

8 METAPOST

This first appears in TugBoat

9 Getting noisy

This first appears in TugBoat

10 Pages

This first appears in TugBoat

11 Flagging

This first appears in TugBoat