# Interactive and real-time typesetting for demonstration and experimentation: ETAP

Didier E Verna

## Interactive and real-time typesetting for demonstration and experimentation: ETAP

Didier Verna

**Abstract**

We present ETAP, a platform designed to support both demonstration of, and experimentation with digital typesetting, interactively, and in real-time.

ETAP provides a GUI which currently focuses on all aspects involved in paragraph formatting. A number of pre-processing features can be switched on or off (hyphenation, kerning, ligaturing, *etc.*). A specific paragraph formatting scheme may be selected from a pool of ready-made algorithms, and adding new algorithms to that pool is easy. Each algorithm comes with its own set of configuration parameters, and the GUI allows you to tweak those parameters and observe the effects in real-time.

ETAP may also be used without, or in parallel with the GUI. While the application is running, the whole programmatic infrastructure is manipulable from a command-line interface. This allows inspection of the various typesetting objects normally displayed by the GUI, and also to perform computations with them, for example, data collection and statistical measurements.

### 1 Introduction

The world of digital typesetting is a fascinating one. As an application domain, it combines a strong focus on aesthetics with many interesting technical challenges, thus making it an Art as much as a Science. The motivation for the project described in this paper is twofold: experimentation (the Science) and demonstration (the Art).

**Experimentation** Suppose you want to try out a new ideas for paragraph justification. Experimentation (including rapid prototyping and debugging) would be made a lot easier with a direct visualization of the results on a sample text (actual contents not necessarily important), and with the ability to interactively tweak such or such parameter from a GUI (Graphical User Interface), while observing the effects in real time.

**Demonstration** In terms of demonstration, my personal experience (in particular when trying to raise students' awareness of the beauty and the subtlety of high quality typesetting) is that showing off static pages of text simply doesn't cut it. On the other hand, there is nothing like having the ability to switch kerning on and off, and immediately see the result, to strike people's minds. The same goes for ligaturing, with no characters actually displayed, but only their bounding boxes which, all of a sudden, go from two or three to just one.

By now, the reader has noticed that whether it is for experimentation or demonstration purposes, the system(s) we are talking about share two common traits: they need to be interactive, and work in real-time. It turns out that, if given those two properties, there is no reason why a single such system couldn't fulfill both objectives. The purpose of this paper is precisely to exhibit one such possible system.

In general, typesetting experimentation is not a very practical thing to do. WYSIWYG (What You See is What You Get) systems are very reactive (for example, you can see the paragraphs being formatted as you type them) but provide neither the highest rendering quality, nor the highest degree of configurability, let alone extensibility. TEX [8, 9], on the other hand, is renowned for the quality of its rendering, but works more like a non-interactive programming language, with its separate development / compilation / visualization phases.

Granted, there are several attempts at bridging the gap. Overleaf, BaKoMa, LyX, and TEXworks provide WYSIWYG environments to TEX, increasing the interactive "feel". Batch Commander [3] was an attempt at providing a GUI for TEX configuration. LuaTEX provides some level of access to TEX's internals. However, none of these systems would let you fundamentally change the way TEX works (they are not meant to).

We must also mention TeXmacs, a very interesting project not in fact based on TEX, but still providing high-quality typesetting from within a WYSIWYG environment. TeXmacs is written in C++ and embeds a Guile interpreter (a dialect of Scheme, from the Lisp family) as an extension language. This makes the project very close to LuaTEX, at least in spirit. This also makes it share the same characteristics: it is a heterogeneous platform, and the C++ core is neither interactive, nor easily modifiable.

As a matter of fact, most available systems today would fail the experimentation goal for a simple reason: they are *production* systems.

We present ETAP (Experimental[1] Typesetting Algorithms Platform), a tool written to ease typesetting experimentation and demonstration. ETAP currently focuses on paragraph formatting, and provides an extensible list of configurable algorithms. ETAP also features switchable kerning, ligaturing, and hyphenation. The source text is editable, and

[1] Whether the "experimental" part refers to typesetting, algorithms, platform, or a combination of them is left to the discretion of the user…

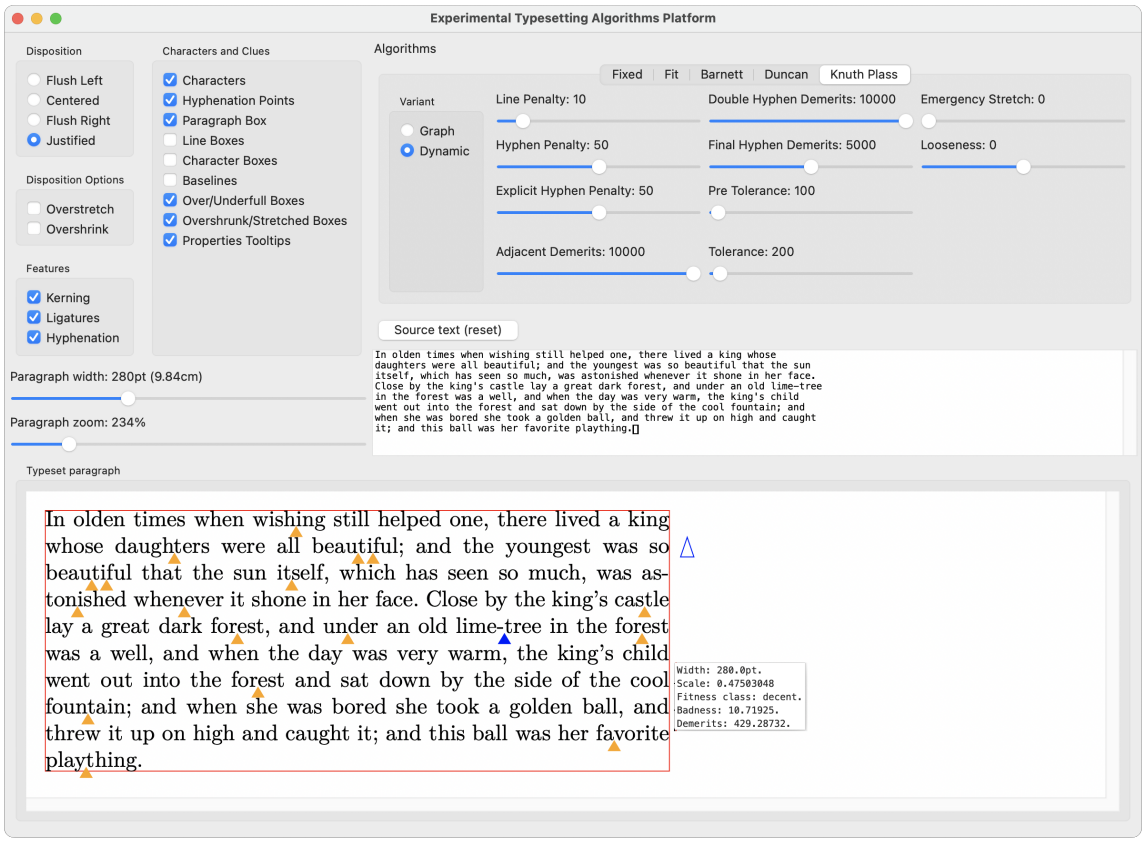


**Figure 1**: The ETAP GUI

the resulting paragraph is (re)displayed in real-time, along with many switchable visual hints, such as paragraph, character, and line boxes, baselines, over/underfull boxes, hyphenation clues, *etc.* All these parameters, along with the desired paragraph width, are adjustable interactively through the GUI.

But ETAP can also be used without, or in parallel with, the GUI, as a scriptable application. This comes directly from its homogeneous design: the application is written entirely in one industrial-strength programming language, Common Lisp [1], which is multi-paradigm, dynamic, and interactive. In particular, fully reflexive access to the various internal data structures, and in fact, to the whole program while it is running, allows for considerable experimentation opportunities, such as batch-formatting in various environmental conditions, and data collection for empirical evaluation and statistical measurements.

Section 2 provides a description of the application's most important features, focusing on interactive manipulation through the GUI. Section 3 discusses some software engineering aspects of its implementation. Finally, Section 4 describes the programmatic (interactive, yet non-graphical) capabilities of ETAP for experimentation.

## 2 The platform

*Note: for the interested reader, another, slightly different description of the platform is available in [16].*

A screenshot of ETAP's GUI is provided in Figure 1. For as much as an interactive and real-time application can be described on paper, the picture should at least give the reader a general feeling of what is available.

The Knuth-Plass algorithm [10] has been selected, along with the default values for all its parameters. The source text for the paragraph is typeset accordingly, in justified disposition, and with kerning, ligatures (although there are none here), and hyphenation.

A number of visual clues have been activated as well and can be observed in the paragraph rendering

area (the bottom half of the window). In addition to the characters themselves, the paragraph's bounding box is drawn. The small arrows pointing upward between characters represent the hyphenation points at which the algorithm has decided *not* to break lines. Finally, the unfilled triangle to the right of line 2 indicates an intentionally overstretched line. This means that the algorithm has decided on a scaling (of glue) ratio which exceeds 1. Indeed, the Knuth-Plass algorithm ran twice here, using a tolerance threshold of 200 the second time. One can also observe that the third line had to be hyphenated, which confirms this is not the result of pass 1 of the algorithm.

Finally, one can see a small popup window near the bottom-right corner of the typeset paragraph. This is actually a "properties tooltip" which pops up when the mouse is moved over a line, and provides feedback on the line in question. In this particular case, it indicates that the line is 280pt wide (the paragraph's width, as the line is properly justified), and is stretched by a scaling factor of approximately 0.475. Also, because the selected algorithm is the Knuth-Plass one, the tooltip reports the line's fitness class, badness, and local demerits. If we were to move the mouse over the paragraph's left margin, the tooltips would advertise a number of global paragraph properties, such as the total demerits, the algorithm's pass number, and the number of remaining active nodes at the end of execution.

Since we are talking about the Knuth-Plass algorithm, note that this project does not aim at providing an exact replica of it, nor of any other currently available line-breaking algorithms (notably Barnett [2] and Duncan [6]), nor of any future ones. In fact, it is our opinion that what is called the "Knuth-Plass algorithm" is actually *not* an algorithm *per se*, but rather the combination of a typical shortest path finding algorithm with a particular cost function having the suitable properties for dynamic programming optimization, all of this written in a relatively low-level imperative language with performance concerns of that time (the 1980s) in mind.

On the other hand, what we are interested in is providing an exact replica of the algorithm's *logic*. Common Lisp is a much higher-level programming language, and most performance concerns of the time have long been obsoleted by the continuously increasing computing power at hand (besides, performance is rarely a top priority for an experimentation platform). Consequently, our design and choice of precise data structures diverge from the original. For example, we actually provide two different implementations of the Knuth-Plass algorithm: one, close to the original, equipped with the same dynamic programming optimization, and another one based on the exploration of a complete graph of solutions (*not* the brute force and exhaustive $2^n$ one, though!).

Another example where we differ from the original is, again, motivated by demonstration and experimentation. In the original Knuth-Plass, pass 1 of the algorithm works on a non-hyphenated text (hyphenation was considered too costly at the time). Only if that fails does TEX hyphenate the text and try a second pass (also with a different tolerance threshold). In our case, we want to be able to display the hyphenation clues every time, if so requested. Consequently, the hyphenation process is implemented as a global option (independent of the selected typesetting algorithm), and pass 1 of the Knuth-Plass algorithm may consequently run on an already hyphenated text, in which case it simply disregards the hyphenation points as potential break points.

## 3 Software engineering

In the context where TEX is still one of the best typesetting systems out there, but also one of the oldest, we deem it important to say a word about software engineering. It is a well-known fact that the science of programming languages and paradigms has evolved considerably over the years. Some people have written about the virtues of a purely functional approach to paragraph breaking in the past [4, 13]. We, on the other hand, favor a more pragmatic than theoretical approach. In particular, instead of having a single paradigm (*e.g.*, functional programming) imposed on us, we prefer the freedom and flexibility provided by a multi-paradigm language [15].

Virtually any programming paradigm aims at increasing both the code's clarity and concision at the same time. Table 1 provides a rough estimate of the project's size in LoC (Lines of Code), and clearly illustrates the benefits of being multi-paradigm for concision. Liang's hyphenation algorithm [11] amounts to 150 LoC. The 500 lines of "lineup" correspond to the pre-processing of the source text, including hyphenation, kerning, ligaturing, and glueing. The currently available paragraph formatting algorithms comprise between 150 and 450 LoC (each variant

**Table 1**: Rough estimate of ETAP's size

| | **LoC** |
|---|---|
| GUI | 800 |
| Hyphenation | 150 |
| Lineup | 500 |
| Algorithms | 150–450 |
| Knuth-Plass | 350 per variant |

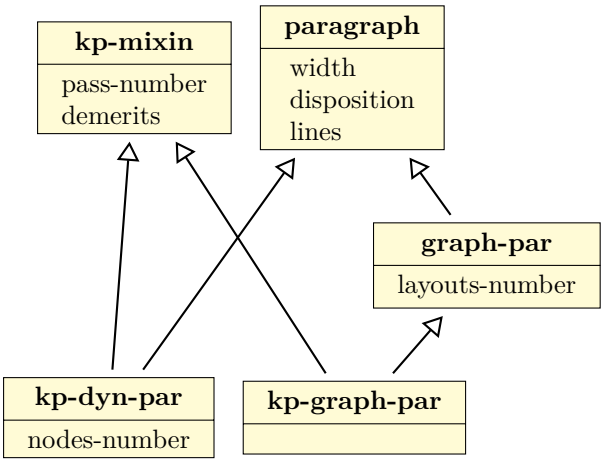


**Figure 2**: The `paragraph` classes

of Knuth-Plass takes 350 lines). We believe this makes the whole platform rather small, considering the functionality offered. In fact, the whole thing currently remains below 5000 LoC, exclusive of large data blocks: an additional 5000 lines is the "lispification" of Adobe's glyph list, and another 5000 lines for English hyphenation patterns.

Let us now illustrate why being multi-paradigm is beneficial for the project, via some examples.

### 3.1 Object orientation

Traditional, class-based object orientation revolves around two fundamental concepts: inheritance (organization of the data) for code reuse, and polymorphism (manipulation of the data) for genericity.

Figure 2 depicts the `paragraph` class hierarchy in a UML fashion. Every subclass incorporates the contents of its superclass(es), thus avoiding duplication. The presence of multiple inheritance (not available in all object-oriented languages) ensures maximum sharing of code: the `kp-dyn-par` class represents paragraphs typeset with the dynamic programming variant of the Knuth-Plass algorithm. Such a paragraph *is* a regular paragraph before anything else, but it also is a `kp-mixin` one, which means that it remembers its pass number and total demerits. Finally, this class also has one additional property of its own: the number of remaining active nodes when the algorithm terminates.

The GUI is passed a paragraph object which, most of the time, is an instance of one of the subclasses (for example, a `kp-dyn-par`). But the visual rendering function is interested only in the contents of the base class (it needs to know only the paragraph's width, disposition and lines to perform the formatting) so it is in fact unaware of the object's exact class. On the other hand, the properties tooltip

```
;; Duncan
(make-graph lineup width)

;; Knuth-Plass, graph variant
(make-graph lineup width
  ;; alternative "next boundaries" function...
  :next-boundaries #'kp-next-boundaries
  ;; ... plus some specific arguments.
  :threshold pre-tolerance)
```

**Figure 3**: The `make-graph` higher order function

popup is implemented via a polymorphic generic function with different implementations for every paragraph class. That is why, in a single function call, it can still advertise the `nodes-number` properties when available, and simply doesn't otherwise.

### 3.2 First class functions

The second example is that of functional programming, although not in the "purely functional" sense mentioned earlier, but rather in Christopher Strachey's sense [5, 14]. Functions in a programming language are said to be "first class", or "first order", or even "higher order" if they behave like any other kind of object: they can be created dynamically, passed as arguments to other functions, provided as return values, *etc.*

Figure 3 provides an illustration of how functional programming contributes to concision as much as object orientation, only in a different way. ETAP has a function called `make-graph` which accepts a lineup and a paragraph width as arguments, and returns a graph of all possible break point solutions. By default, starting at a specific position in the lineup, the next possible break points would be those involving a scaling of at most 1 in absolute value. There is a function called `next-boundaries` which computes the list of such break points.

On the other hand, some algorithms may have a different view on what the next possible break points actually are. For instance, the Knuth-Plass algorithm does not look at the scaling alone, but considers hyphenation and uses a pre-tolerance or a tolerance threshold, depending on the pass number. Creating a Knuth-Plass graph thus only differs from a regular one in the way the next possible break points are computed. It would be unsatisfactory to write a specific version of `make-graph` just because of that small divergence from the default behavior. In fact, most of the code would actually be redundant with the regular version.

What we do instead, is parameterize the "next boundaries" function. The Knuth-Plass implementation comes with an alternative function called `kp-next-boundaries`. As you can see in Figure 3, `make-graph` is in fact a higher order function, accepting a "next boundaries" function as argument. This ensures that the skeleton of `make-graph` does not need to be duplicated.

## 4 Experimentation

The last critical software engineering aspect which we want to emphasize is the dynamic and interactive nature of Common Lisp, ETAP's implementation language. Just like the more mainstream scripting languages such as Ruby, Python, or Perl, Lisp provides a REPL (Read Eval Print Loop) from which the programmer can interact with the program while the program is running. In fact, both the REPL and the GUI may be used at the same time to interact with the system, and the homoiconic [7, 12] nature of the language makes it trivial to introspect the live objects, or even destructively modify them.

A typical experimentation scenario is as follows. The programmer runs a typesetting experiment via the GUI in various conditions, and observes a surprising (or suspicious) situation. The programmer then switches to the REPL and from there, has the ability to inspect (or debug) the complete program state, without leaving the program. It is even possible to hot-modify the typesetting code, for example to fix a bug and switch back to the GUI in order to trigger a redisplay.

Let us now illustrate the benefits of interactivity with two examples, the second being a recent and true anecdote.

### 4.1 Statistics

ETAP provides a short (around 200 LoC) generic infrastructure for data collection and statistical measurements of all sorts. For example, there is a function called `scalar-statistics` that loops over all available algorithms and paragraph widths, and each time collects a scalar value computed by a function passed as an argument (another case of functional programming at work).

This function can be used to generate comparative charts for any criterion one may think of. For example, with the two function calls below, we are able to generate the charts presented in Figures 4 and 5.

```
(scalar-statistics #'collect-scales-mean)
(scalar-statistics #'collect-scales-variance)
```

Those charts are primarily meant to be visualized on (large) screens, so they will appear somewhat cluttered in a PDF. Their actual content is not so important here, as the point is merely to illustrate the current capabilities of the platform.

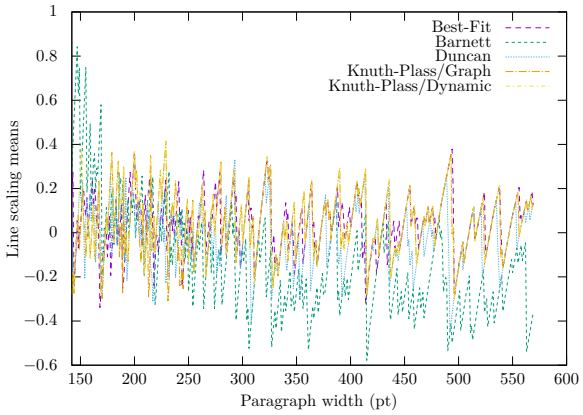


**Figure 4**: Scales mean

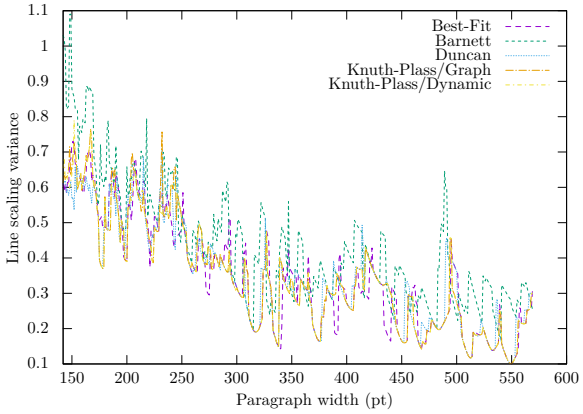


**Figure 5**: Scales variance

The first one shows the average line scaling for every algorithm and paragraph width. It is immediately visible on this chart that except for very narrow paragraphs, the Barnett algorithm has a tendency to compress a lot. On the other hand, the second chart also shows that Barnett has a higher scaling variance than the other algorithms. In TEX's terms, this means that the adjacent demerits would probably be off the charts (and that would be easy to confirm too).

These two charts are just examples. Other ready-made data collection functions allow you to compute the graph sizes, the number of possible solutions, the number of under/overfull lines, *etc.*, usually in less than 15 LoC.

### 4.2 The anecdote

The final example we want to provide here takes the form of an anecdote, and we think it illustrates pretty

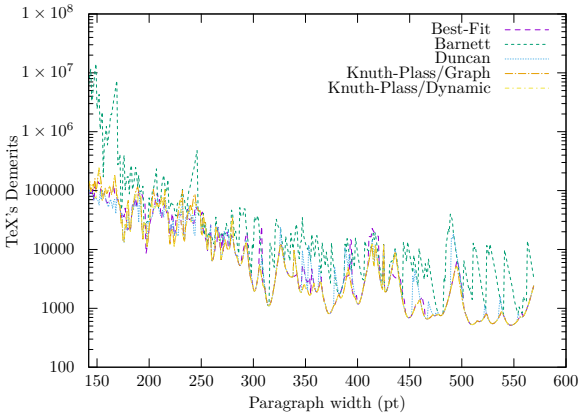


**Figure 6**: TEX's view of the competition

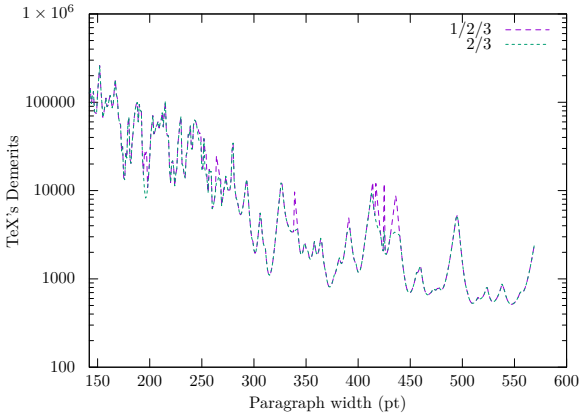


**Figure 7**: With or without pre-tolerance

well why having such a platform for experimentation is convenient.

At some point, we were curious to get an idea of TEX's view of the competition. In other words, the question was: given a paragraph formatted by an algorithm other than Knuth-Plass, what is this paragraph's number of total demerits?

We thus wrote a 50-line function to compute the chart presented in Figure 6. Of course, the expected general result is that when there is a valid paragraph breaking solution, TEX should find itself better than the competition, because again, it *does* find optimal solution according to its own quality criteria. Figure 6 does confirm this, although if you look closely, you will spot a curious area, near the 440pt paragraph width, where the Best Fit algorithm seems to perform better.

At first, we thought there was a bug somewhere in the implementation of one algorithm or the other, but in fact the code was correct. Visualizing the resulting paragraphs made it apparent that the Best Fit solution contained hyphens, and the Knuth-Plass one didn't. Recall that TEX will stop at pass 1 of the algorithm if it finds a valid solution without hyphens.

The next hypothesis, then, was that there would be cases in which a hyphenated paragraph would amount to fewer demerits than its non-hyphenated counterpart. Once you come to think of it, this hypothesis is in fact very plausible, given that by default, hyphen penalties are quite small (50) compared to, say, adjacency penalties (10000).

We were able to confirm that hypothesis in literally one line of code. Indeed, generating the chart presented in Figure 7 took only one function call. This chart plots the demerits for all paragraph widths with or without pass 1 of the Knuth-Plass algorithm. Recall that short-circuiting pass 1 is done by setting the pre-tolerance parameter to $-1$ (which is what LATEX does, by the way).

On this chart, some areas are clearly visible where pass 2 of the algorithm performs better (in terms of total demerits) than pass 1. It is a bit surprising that after 25 years of using LATEX, we only recently realized that. But the important point, here is how ETAP made the experimentation, hypothesis formulation, and confirmation simple.

## 5 Conclusion and perspectives

As we hope to have demonstrated in this paper, ETAP has now reached a state where it is suitable for both demonstration and experimentation. The project is available on GitHub (`github.com/didierverna/etap`), and as a matter of fact, we were quite happy that after the presentation at TUG 2023, half a dozen attendees immediately expressed some interest in using it. Consequently, we immediately updated the installation instructions so that everyone may now use it, without any prior knowledge of Lisp.

The existing list of planned improvements is already quite large. For example, more flexibility in font selection is a high priority. In general, our plans for the future will follow three complementary directions.

**Bibliography** We plan on studying more line-breaking literature and port existing ideas or algorithms we find to ETAP. By the way, here is a small plea for help: our Duncan [6] and Barnett [2] implementations are based only on the descriptions that are given in the Knuth-Plass paper. We couldn't recover the original publications, and hence would love it if anyone could contribute them.

**Research** One of our top priorities in the research area is working on river detection. We also plan on

looking into microtype extensions, and perhaps a number of smaller or simpler issues.

One such issue is what we call "character ladders". In the same way TEX has this notion of "double hyphen demerits" for hyphenation ladders, it is usually undesirable that consecutive lines begin or end with the same characters or short words. Taking this into account is in fact very simple, and can even be implemented as an extension to the Knuth-Plass algorithm by adding a new kind of demerits.

Speaking of Knuth-Plass extensions, the ability to have a graph-based implementation as well as the original dynamic programming optimization opens the door to a number of interesting research questions. For example, the way TEX addresses adjacency problems is rather coarse. It only has four fitness categories because in order for its cost function to remain dynamically optimizable, the number of active nodes to keep around depends on the number of fitness classes (which needs to be discrete!). This is in fact sub-optimal because if two consecutive lines belong to different classes, the adjacency cost will be the same, whether or not those lines are close to each other in terms of scaling. On the other hand, if one maintains a full graph of possible breaking solutions, the adjacency demerits can be turned into a continuous, hence much more accurate function. It would be interesting to see how much of a difference this makes in practice.

**Development** Finally, there are also some more technical aspects that we want to address, one of them being a tighter integration between the GUI and the platform's core. The current design offers a number of features that are not yet accessible to the GUI. For example, every possible break point in the lineup has a local penalty value that can be changed programmatically. The GUI only allows global changes to the default initial value (TEX's hyphen penalty for example), but it would be nice if we could, say, right click on hyphenation points, and get a slider to change said penalty, having the paragraph reformatted immediately. Indeed, this would be an extremely convenient feature to have in a production system as well.

## References


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

[2] M.P. Barnett. *Computer Typesetting: Experiments and Prospects*. MIT Press, Jan. 2000.

[3] K. Bazargan. Batch commander: a graphical user interface for TEX. *TUGboat* 26(1):74–80, 2005. tug.org/TUGboat/tb26-1/bazargan.pdf

[4] R.S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming* 6(2):159–189, 1986. doi.org/10.1016/0167-6423(86)90023-7

[5] R. Burstall. Christopher Strachey — Understanding programming languages. *Higher Order Symbolic Computation* 13(1–2):51–55, 2000.

[6] C. Duncan, J. Eve, et al. Computer typesetting: an evaluation of the problems. *Printing Technology* 7:133–151, 1963.

[7] A.C. Kay. *The Reactive Engine*. Ph.D. thesis, University of Utah, 1969.

[8] D.E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[9] D.E. Knuth. *TEX: The Program*, vol. B of *Computers & Typesetting*. Addison-Wesley, Jan. 1986.

[10] D.E. Knuth, M.F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience* 11(11):1119–1184, 1981. doi.org/10.1002/spe.4380111102

[11] F.M. Liang. *Word Hy-phen-a-tion by Com-put-er (Hyphenation, Computer)*. Ph.D. thesis, Stanford University, 1983. tug.org/docs/liang

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

[13] O. de Moor, J. Gibbons. Bridging the algorithm gap: a linear-time functional program for paragraph formatting. *Science of Computer Programming* 35(1):3–27, 1999. doi.org/10.1016/S0167-6423(99)00005-2

[14] J. Stoy, C. Strachey. OS6 — An experimental operating system for a small computer. Part 2: Input/output and filing system. *The Computer Journal* 15(3):195–203, 1972.

[15] D. Verna. Star TEX: the next generation. *TUGboat* 33(2):199–208, 2012. tug.org/TUGboat/tb33-2/tb104verna.pdf

[16] D. Verna. ETAP: Experimental typesetting algorithms platform. In *15th European Lisp Symposium*, pp. 48–52, Porto, Portugal, Mar. 2022. doi.org/10.5281/zenodo.6334248



◇ Didier Verna
EPITA Research Lab
14–16, rue Voltaire
94270 Le Kremlin-Bicêtre
France
didier (at) lrde.epita.fr
https://www.lrde.epita.fr/~didier/
ORCID 0000-0002-6315-052X