Baskerville

The Annals of the UK TEX Users' Group **Guest Editor: Kaveh Bazargan** **Vol. 7 No. 1**

ISSN 1354–5930 **September 1997**

Articles may be submitted via electronic mail to `baskerville@tex.ac.uk`, or on MSDOS-compatible discs, to Sebastian Rahtz, Elsevier Science Ltd, The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, to whom any correspondence concerning *Baskerville* should also be addressed.

This reprint of *Baskerville* is set in Times Roman, with Computer Modern Typewriter for literal text; the source is archived on CTAN in `usergrps/uktug`.

Back issues from the previous 12 months may be ordered from UKTUG for £2 each; earlier issues are archived on CTAN in `usergrps/uktug`.

Please send UKTUG subscriptions, and book or software orders, to Peter Abbott, 1 Eymore Close, Selly Oak, Birmingham B29 4LB. Fax/telephone: 0121 476 2159. Email enquiries about UKTUG to `uktug-enquiries@tex.ac.uk`.

Contents

1 Editorial

We had fun trying to output this issue together. Because of the very nature of Baskerville, being written by the TEXperts, and pushing TEX to its limits, it is not the simplest publication to handle. Now we all like to do things in the most elegant way possible but, having run a production environment for a few years, we have learned to use the quick and dirty method when it works. I thought some notes on the production of this issue might be of interest.

The work was done using Textures on a Power Computing Macintosh clone. The initial problem came from (of all places) fonts. Textures uses a system of font management different to other machines (in itself no bad thing). A version of the Baskerville class file had to be chosen from the several that were offered/located (actually a `diff` was performed on them and the most likely looking candiates were pinched into one big, mutated, class file—it worked, so who are we to complain). Things went smoothly until Sebastian's MetaPost article where the use of `%*Font`, dvips specific, commands meant that the `.eps` files needed to be modified to run under Textures. The following lines where added. (This is not a general solution and can be quite dangerous, but it was quick.)

```
/cmsy10 /CMSY10 def
/cmr10  /CMR10  def
/cmmi10 /CMMI10 def
/fshow {exch findfont exch scalefont setfont
        show}bind def
```

What it does is to make the text `cmr10`, etc active (so no phrases like 'this figure uses cmr10' are allowed) and defines the `fshow` command. Unfortunately you must have access to *all* the fonts locally on your machine as they aren't embedded into the PostScript. Obviously we could have run MetaPost on the original source codes, by adding a `prologues := 1;` command of course, but as both methods require us to change all the files and the latter requires an extra processing step from METAPOST so the first method was deemed acceptable.

Jeff Kingston is the author of the batch-processing document formatter lout. His paper was written while he was on sabbatical in the UK, and he was happy for it to be republished in *Baskerville* (after translation to LATEX, of course). The paper is over a year old, but Jeff welcomes comments (other than those of the form "if you did this in LATEX, all your problems would go away"...).

The paper on standard DTDs was first published in EPSIG News 5 number 3, September 1992, pp 10–19. Permission was obtained from the three authors to republish in *Baskerville*.

Despite the extreme age of this paper it contains much useful comment and observation of the problems of encoding mathematical notations. Two appendices have been omitted: 'Existing mathematical notations' and 'Comparison between ISO TR 9573 and AAP Math DTDs'.

Good luck to the next editor!

I The Future of Document Formatting (Working Paper)

1 Abstract

Document formatting systems have reached a plateau. Although existing systems are being steadily enhanced, the next major step forward will require a union of the best features of batch formatters, interactive document editors, and page description languages. This paper draws on its author's twelve years of experience designing, implementing, and enhancing the Lout document formatting system to identify the remaining problems in document formatting and explore some possible solutions.

2 Introduction

Document formatting is one of the most widespread applications of computers. Improvements in document formatting software and the hardware on which it is based have revolutionized the production of documents and enlarged our conception of what a document might be.

Any attempt at this point to define 'document' would run a risk of being overtaken by events; already documents commonly include moving images, sound, and dynamic updating as their sources of information change in real time. It is perhaps safe to say that a document is information arranged for presentation to a person; the information may be called the *content*, and the arrangement its *layout*. Document formatting is essentially about mapping content to layout, although functions that do not exactly fit this definition, such as spelling and grammar checking, or even creation and editing of content, are often found in document formatting systems.

Document formatting systems fall into two camps. In one camp are the interactive document editors, ranging from word processing systems such as Microsoft Word [18] up to desktop publishing systems such as FrameMaker [2] and Interleaf [9]. These offer an editable screen image of the document layout. In the other camp are the batch formatters, such as troff [19], Scribe [21], TEX [15], and Lout [13], which process text files with embedded markup to produce non-editable layout. In this paper the above names will stand for the entire software family; TEX includes LATEX [17], FrameMaker includes FrameMaker+SGML, and so on. Somewhere in between are the hypertext [8] net browsers, based on HTML, which are primitive batch formatters offering limited interactivity such as the ability to click on a hyperlink or fill in a form.

All of these systems are being actively enhanced by their developers, with new versions appearing regularly. For example, FrameMaker and Interleaf have responded to the World-Wide Web phenomenon by adding support for SGML [7] and HTML. Nevertheless, viewed from a wider perspective, they all appear to have reached a plateau, in the sense that each has fundamental limitations that are not likely to be overcome. For example, troff, TEX and Lout are batch formattersand are not likely to become interactive; FrameMaker and Interleaf are not as extensible as the batch formatters and, again, are not likely to become so.

One frequently hears arguments for or against these systems, but the truth is that none of them is ideal yet all have something to offer to the future of document formatting. What is needed now is a synthesis of the best features of all of these systems.

Papers which reflect on document formatting seem to be very rare. The survey paper by Furuta, Scofield and Shaw [6] is still well worth reading; Kernighan [11] reflects on the troff family; this author has described the design and implementation of Lout [12]. But for the most part one has to infer principles from the systems themselves, and to look among the specialized applications such as music formatting [5], graph drawing [10, 23, 16], or non-European languages for requirements.

This paper draws on its author's twelve years of experience in designing, implementing, and enhancing the Lout document formatting system, plus his more limited experience of the systems mentioned above, to identify a set of requirements for a document formatting system that would be a significant advance on all current systems, and to explore their interactions.

3 Requirements

This section identifies the most significant requirements for a document formatting system. Efficiency in space will cease to be a requirement in the next few years. Efficiency in time is of course essential, as are other requirements that apply to any large software system, such as robustness, openness, and an interface that permits users of varying levels of expertise to work productively.

The other requirements are editability, extensibility, generality, and optimality. Each of these requirements is discussed in turn in the sections that follow, together with problems that it presents either alone or in conjunction with previous requirements.

It is not possible to prove that this list of requirements is complete, but the author has carefully compared it against the features of most of the document formatting systems listed earlier. The only major omission has been the convenience features commonly found in interactive systems, such as spelling and grammar checkers, input and output in a variety of data formats, version control, and so on. These are valuable features, but they have little to do with document formatting in the core sense of mapping content to layout.

3.1 Editability

Editability, the ability to edit content while viewing layout, is the strong suit of word processing and desktop publishing systems. Fairly or not fairly, many users will not accept batch formatting. Also, the batch formatting edit-format-view cycle is too slow when the layout rule is 'what pleases the eye,' such as in diagrams, or when content must be altered to achieve a good layout, for example in paragraphs containing long unbreakable inline equations.

Interactive interfaces also have an advantage when the logical structure does not follow a tree pattern. A good example is the editing of graphs (the combinatorial kind). Users of an interactive system can click on any pair of nodes to indicate that they are to be joined by an edge. In a batch system, because the structure is not tree-like, it is necessary for the user to invent names for the nodes and use the names when creating edges, which is considerably more error-prone. By contrast, equations do follow a tree pattern and so there is never any need to attach names to subexpressions.

Critics of interactive systems typically complain about the lack of content structure in interactive editors, and also about their weakness as editors compared with good text editors. Neither problem would seem to be inherent, and in fact recent versions of high-end document editors (FrameMaker+SGML for example) are addressing the content structure problem.

Openness to such auxiliary applications as free-text search and retrieval and creation of documents by computer programs requires that an archive format based on marked-up text be included in any interactive system. It only takes a little care to make such a format readable by humans. Thus an interactive system is automatically also a batch system.

3.2 Extensibility

Extensibility in a document formatting system means the easy addition of new features. It is the strong suit of batch formatters. For example, this author's Lout system has no built-in knowledge of equations, tables, or page layout (not even the concept of a page is built-in); these are all added by means of packages of definitions written in the Lout language, which is sufficiently high-level to make them fairly easy to produce.

Extensibility implies some initial kernel of primitive features upon which the extensions are built. These would include horizontal and vertical stacking, rotation, and so on. The most interesting such feature is the mechanism for getting floating figures and footnotes into their places: diversions and traps in troff, floating insertions in TEX, galleys in Lout. There must also be ways of combining and packaging the primitives into features useful to the end user.

Although a system not built on such a kernel is conceivable, it seems scarcely possible to this author that such a system could supply all the features demanded by end users. The list is so vast – equations, tables, graphs, chemical molecules, music, and so on – that some kind of high-level kernel language seems essential to achieving them in any reasonable time and with any consistency, just as high-level programming languages are essential to large software projects.

Typography generates requirements for many features, such as hyphenation, spacing and kerning, ligatures, and so on. A document formatting system must produce good typography, because end users cannot be expected to do it themselves. Many of these features are dependent on the current language, and many English or European-oriented systems have failed to be extensible to the typography of languages outside that sphere. A good source of features needed in world-wide typography is Apple Computer's QuickDraw GX [3], although their approach of implementing the features in C is relatively non-extensible since it requires recompilation.

When an interactive system is extended with a new feature, it must be possible to continue editing in its vicinity.

Ultimately, the layout of a document is a function of its content, so we may identify features with functions. In

extreme cases, such as optimal layout, a function may take the entire document as its parameter; but usually it has small, clearly delimited parameters as in

$$\mathit{built\_up\_fraction}(\mathit{numerator}, \mathit{denominator})$$

There may also be implicit parameters inherited from the context, such as the current font size.

It is quite reasonable to insist that within any editing session the collection of features be immutable. Thus it is not essential to be able to edit the definition of any function while viewing any layout. In some cases, such as simple abbreviations, editing of definitions is quite simple and could easily be supported. But more complex functions, such as optimal layout or graph layout, are defined by computer programs and so are not amenable to editing in this way.

In a similar vein, it is correct to insist that those parts of the layout originating within definitions be immutable. For example, the bar in a built-up fraction should not be editable. This does not preclude the addition of parameters to *built_up_fraction* to control the appearance of the bar if desired, but to allow the user to arbitrarily change the bar would produce a layout whose origin as a built-up fraction must be lost.

Thus, editability of features really only means editability of their parameters.

The most favourable case occurs when the function displays a parameter in a form similar to that which it would have taken if it had been entered outside the function. For example, *built_up_fraction* displays both its parameters, changing their appearance only slightly (by squeezing vertical spacing within them, and possibly changing the font size). The user can edit such a parameter as though it was not a parameter at all, and so (inductively) can edit parameters of parameters and so on without limit. This is essentially how equation editors work, and the Lilac system [4] has demonstrated it in an extensible framework, although using a kernel language too incomplete to support the full range of features required by users. A function may display a parameter more than once, in which case editing one display must change them all.

Preserving editability of displayed parameters is a difficult problem when the function is implemented externally to the document editing system. For example, if an external graph layout program [16] is employed, the result cannot be returned as a bitmap or PostScript file; rather a set of coordinate pairs or something similar is required so that the document formatter can place the nodes itself and hence understand where they ended up.

It has been suggested that a non-editable result is acceptable in such cases if a click in the region it occupies signals the opening of a separate editor that does undertand what is going on in that region. This is the interactive equivalent of the preprocessor approach used by troff, and it has the same drawbacks of lack of consistency, duplication of features, and loss of generality (since even if every editor may invoke every other editor, the communication channels between them typically cannot convey such information as the current font, available space, and so on). An architecture based on a single master editor with slave non-interactive formatting programs is preferable.

Parameters which are not displayed are a nightmare, and are responsible for much of what is ad-hoc in existing interactive systems. Two main approaches are in use. The first is the ‘style sheet’ or ‘dialogue box’ approach, in which the user who selects a feature with non-displayed parameters is presented with a box listing them and asked to supply values: a font name, a location to place a figure, a style of numbering, or whatever. This is the most general method, easily adapted for use in an extensible system. It works particularly well when the parameters have sensible default values, for then use of the box is optional, and when they have only a small range of possible values, for then the values may be displayed in a menu.

Second is the ‘inference’ method. Every parameter has some effect on layout, otherwise it would be useless. So the user is offered a means of manipulating layout, and the parameter’s value is inferred from it. For example, most editors permit an included graphic to be clipped by clicking on its boundary and moving the mouse; scaling and even rotation may be set by such means. Drawing programs allow nodes to be dragged about in the drawing area. ‘Master pages’ or ‘template pages,’ which allow the user to specify entire page layouts involving many parameters simultaneously, demonstrate the value of the inference method.

The great drawback of the inference method is that an inference interface has to be invented for every non-displayed parameter, and this is difficult in an extensible system. However, it should at least be possible to implement an inference interface for all suitable non-displayed parameters of kernel features, such as the *boundary* parameter of *clip*(), and in cases such as

$$\begin{gathered}\mathit{define\ user\_level\_feature}(\ldots, \mathit{boundary}, \ldots) = \\ \ldots \mathit{clip}(\ldots, \mathit{boundary}, \ldots) \ldots\end{gathered}$$

to propagate this interface upwards from kernel features to user level features. Then every user level feature that offers clipping as a parameter, for example, will do so in the same way.

3.3 Generality

By generality we will mean the absence of illogical restrictions on the use of features, either in the contexts in which they may be used, or in the values that may be assigned to their parameters. (These are formally the same thing, but the distinction is useful.)

Examples of illogical context restrictions are extremely common in document formatting systems. FrameMaker permits objects to be rotated in certain contexts (when they are table entries, for example) but not others. In troff it is very easy to include an equation within a table, but very much harder to include a table in an equation. Not all context restrictions are illogical, of course: a chapter should not begin within a table, for example.

Lack of context generality takes a severe toll, because it means that implementation code, possibly highly sophisticated and with a great deal to offer, is locked into a few limited contexts. For example, FrameMaker has a very interesting equation editor, but there seems to be no hope that its code can be used for such tasks as editing tree diagrams or diagrams of chemical molecules, despite the technical similarities among these tasks.

Examples of illogical domain restrictions are particularly common among geometrical functions. For example, LaTeX will produce lines only at certain fixed angles, and most systems only really understand rectangular shapes. The PostScript page description language [1] is far ahead of everything else in geometrical generality: in PostScript, arbitrary curves (even disconnected ones) made of lines, arcs, and Bezier curves may be drawn and filled, and arbitrary combinations of rotation, scaling and translation may be applied to arbitrarily complex fragments of documents lying within one page.

The abandonment of rectangles in favour of arbitrary shapes would have widespread beneficial effects if done in full generality. Text could fill arbitrary shapes and run around arbitrary graphics. Fonts could be defined (as they are in PostScript) as collections of arbitrary shapes, permitting kerning of arbitrary pairs of glyphs, not just glyphs of equal font and font size as at present, thus solving the subscript kerning problem. Line spacing could reflect the true appearance of lines, not be crudely based on the highest ascender and lowest descender. Optimizations based on bounding boxes and caching should be able to solve the efficiency problems.

3.4 Optimality

By optimality is meant the ability to find the best possible layout for the given content. An optimal layout is not necessarily a good layout, because some documents have no good layout. Optimal layout thus cannot remove the burden of rewriting content to achieve good layout, but in practice it does greatly reduce that burden, and this is why it is has been included.

The idea that layout could be optimal seems to be due to Knuth and Plass [14], who presented an algorithm for the optimal breaking of a paragraph into lines which is used in Knuth's TeX system. Research work was done on more general optimality as well [20], although this author is unsure how much of this work was incorporated into TeX.

Suitably generalized, their paragraph breaking algorithm is as follows. The first step is to deduce from the content a sequence of atomic formatting steps. For example, the content

The cat sat on the mat

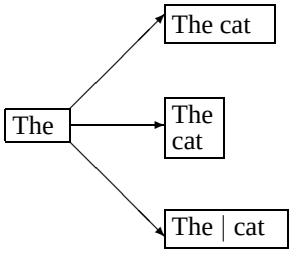
might have sequence

create_empty_paragraph
add_word_to_paragraph(*The*)
add_word_to_paragraph(*cat*)
...

Every prefix of this sequence should define a legal document in its own right; the whole sequence defines the document we wish to format. The question as to what constitutes an atomic operation is not of fundamental importance; one could choose to add one letter at a time, or an entire paragraph.

Define a *badness* function from layouts to integers. Small values indicate good layouts, large values indicate poor ones. There are no restrictions on how this function is defined, except the practical one of being computable in a reasonable time.

Now there will be several ways in which each atomic step may be performed. For example, *add_word_to_paragraph* could add its word to the end of the current line, or it could start a new line, or it could even start a new page or column. This leads to a tree structure:



Each node is a layout of a partial document, each edge is one atomic operation.

The next atomic operation is applied to each leaf node, creating more partial documents, and so on until the sequence ends and the leaf nodes represent all layouts of the document of interest. The leaf node of minimum badness is the optimal layout.

This model can incorporate diverging operation sequences caused by layout dependencies. For example, suppose the word *abacus* has an index entry attached to it, and that along one path in the tree this word appears on page 99, while along another it appears on page 100. Then, in the sequence of operations defining the index, we will find

$$\begin{array}{l}\ldots\\ \mathit{add\_word\_to\_paragraph}(\mathit{abacus})\\ \mathit{add\_word\_to\_paragraph}(\mathit{99})\\ \ldots\end{array}$$

along one path, and

$$\begin{array}{l}\ldots\\ \mathit{add\_word\_to\_paragraph}(\mathit{abacus})\\ \mathit{add\_word\_to\_paragraph}(\mathit{100})\\ \ldots\end{array}$$

along the other. However, forward references create cyclic dependencies which cannot be handled in this way. For them, it seems to be necessary to add operations which change the value of words that have already been laid out, and to propagate the resulting changes until they die out. In rare cases this method will cycle forever, but in practice it is probably not difficult to avoid this problem using tricks such as refusing to allow a revision to reduce the number of lines allocated to a paragraph.

The algorithm as expressed has exponential time complexity. In practice, however, the number of different layouts of a document that are close enough to optimal to deserve examination is likely to be quite small. The challenge, then, is to find ways to prune the layout tree severely while retaining enough of it to discover, for example, that setting a sequence of paragraphs tight or loose will avoid a bad page break further on. This is an area needing detailed research; we can only glance at a few obvious possibilities here.

If the badness function is monotone increasing along every operation sequence, then a bad node can only have worse successors, and this justifies pruning its entire subtree. Monotonicity is not guaranteed (for example, adding one word to a paragraph which has a widow word will reduce its badness) but it is probable that tricks such as ignoring widow words in incomplete paragraphs can bring us near enough to monotonicity to justify pruning bad nodes.

One immediate application is to prune nodes whose layouts are obviously terrible, such as nodes containing clearly premature line endings or page endings. Indeed, it should be possible to avoid even generating such nodes.

When it can be established that two nodes are equivalent, in the sense that they lay out the same subsequence and their layouts occupy the same space, their future careers must be identical and the worst of the two may be pruned. The tree structure becomes a graph, and the optimal layout algorithm may be viewed as a shortest path algorithm, as described by Knuth [15].

Establishing the equivalence of two nodes may not be easy. There certainly is not time for complex comparisons of all pairs of layouts of a given subsequence. Knuth and Plass's algorithm recognises that two nodes are equivalent when they lay out the same subsequence and the most recent choice on the path to each was to start a new line. This same idea may be used to equivalence all paths into one at the new-page operation preceding a new chapter.

Another useful idea is to group operations together, find optimal layouts for the group separately, then introduce an atomic operation at a higher level which represents the entire group. Grouping the operations that define one paragraph in this way is very beneficial, for example. In isolation, optimal pragraph breaking explores many options, but in the

end it is likely to return only at most two reasonable distinct results, of n and $n+1$ lines respectively for some n, and these become the only choices for the atomic *add_paragraph* operation that represents the whole group at the higher level. Furthermore, these two results may be cached and used without recalculation on every path containing that particular *add_paragraph* operation whenever the margins have the same width.

With care, suppressing tiny variations introduced by ascenders and descenders on letters, the layout tree might be induced to contain only as many paths as the difference in the total number of lines between the loosest and tightest settings of the paragraphs inserted so far, and over the course of one chapter this might be a manageable number. For safety, a fixed upper limit could be placed on the number of nodes kept, producing a beam search [22] which would definitely bound the time complexity to a fixed multiple of the cost of non-optimal layout, while sacrificing guaranteed optimality.

There do not seem to be any extra problems in incorporating optimality into an extensible system. Users would certainly welcome options to user-level features such as 'insert this figure either following the current line, or at the top of the next page, whichever looks best.' Whether an editable system can offer optimal layout without exceeding response time bounds is a matter for further research. There should be time to maintain optimality of the current paragraph at least, and if the current chapter is set within constant-width margins, it should be no more time-consuming to maintain optimal layout in a twenty page chapter than it is in a twenty line paragraph, provided the two alternative paragraph breaks of each non-current paragraph of the chapter are cached. If the cost does prove too great, optimality could be relegated to a button that the user can press just before going for coffee.

4 Conclusion

This paper has demonstrated that a next-generation document formatting system, incorporating the best features of current systems in full generality, is neither logically inconsistent nor likely to be infeasibily slow.

The major design problem is the identification of a suitable kernel of primitive features. Given the massive superstructure that this kernel will support, its design quality must be of the highest. This design was not attempted in this paper, but the author believes that the kernel of the Lout document formatting system would make a good starting point, although it is too incomplete, insufficiently general, too large, and occasionally too imprecisely defined to serve as the kernel of a next-generation system as it stands.

The major implementation problem is to find optimizations that preserve generality yet achieve the required response time. This paper has pointed out optimizations that seem quite likely to be adequate on hardware that will be widely available in a few years.

It is also to be hoped that next-generation systems will finally lay to rest the language issues that bedevil systems created within an English or European language framework. Given sufficiently general primitives, this should be an easy matter.

5 Acknowledgements

The author gratefully acknowledges comments on the first draft of this paper received from Mike Dowling, Ted Harding, Robert Marsa, and Basile Starynkevitch.

References

[1] Adobe Systems, Inc. *PostScript Language Reference Manual, Second Edition*. Addison-Wesley, 1990.

[2] Adobe Systems, Inc. *Using FrameMaker+SGML*. Adobe Systems, Inc., 1995.

[3] Apple Computer, Inc. *Quickdraw GX*. 1996. Available as `http://support.info.apple.com/gx/gx.html`

[4] Kenneth P. Brooks. Lilac: a two-view document editor. *IEEE Computer*, pages 7–19, 1991.

[5] Eric Foxley. Music — a language for typesetting music scores. *Software—Practice and Experience*, 17:485–502, 1987.

[6] Richard Furuta, Jeffrey Scofield, and Alan Shaw. Document formatting systems: survey, concepts, and issues. *Computing Surveys*, 14:417–472, 1982.

[7] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990. ISBN 0-19-853737-9.

[8] Charles F. Goldfarb. Hytime: a standard for structured hypermedia interchange. *IEEE Computer*, 24:81–84, 1991.

[9] Interleaf, Inc. *Interleaf 6 for Motif: next generation document creation, composition and assembly*. 1996. Available as `http://www.interleaf.com/i6motifds.html`

[10] Brian W. Kernighan. Pic — a language for typesetting graphics. *Software–Practice and Experience*, 12:1–21, 1982.

[11] Brian W. Kernighan. The unix system document preparation tools: a retrospective. *AT&T Technical Journal*, 68:5–20, 1989.

[12] Jeffrey H. Kingston. The design and implementation of the lout document formatting language. *Software–Practice and Experience*, 23:1001–1041, 1993.

[13] Jeffrey H. Kingston. *The Lout Document Formatting System (Version 3)*. 1995. Available as `ftp://ftp.cs.usyd.edu.au/jeff/lout/`

[14] D. E. Knuth and M. E. Plass. Breaking paragraphs into lines. *Software–Practice and Experience*, 11:1119–1184, 1981.

[15] Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.

[16] Balachander Krishnamurthy, editor. *Practical Reusable UNIX Software*. John Wiley, 1995.

[17] Leslie Lamport. *LATEX User's Guide and Reference Manual*. Addison-Wesley, 1986.

[18] Microsoft, Inc. *Microsoft Word*. Microsoft, Inc., 1996. Available as `http://www.microsoft.com/msword/`

[19] Joseph F. Ossanna. "nroff/troff" user's manual. Technical Report 54, Bell Laboratories, Murray Hill, NJ 07974, 1976.

[20] Michael F. Plass. *Optimal pagination techniques for automatic typesetting systems*. PhD thesis, Stanford, CA, 1981.

[21] Brian K. Reid. A high-level approach to computer document production. In *Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL), Las Vegas NV*, pages 24–31, 1980.

[22] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition edition, 1992.

[23] Christopher J. Van Wyk. *A language for typesetting graphics*. PhD thesis, Stanford, CA, 1980.

II Standard DTDs and scientific publishing

N. A. F. M. Poppelier (n.poppelier@elsevier.nl),
E. van Herwijnen (eric@vanherwijnen.org), and
C.A. Rowley (C.A.Rowley@open.ac.uk)

1 Abstract

This paper has two parts. In the first part we argue that scientific publishing needs one standard DTD for each class of documents that is published. For example one for all research papers and one for all books. In the second part we apply this reasoning to mathematical formulas, and we outline some design requirements for a document type definition for mathematical formulas. In the appendices we discuss and compare existing document type definitions for mathematical formulas.

2 Introduction

In the preface to [1] Charles Goldfarb wrote that the Standard Generalized Markup Language can be described as many things, and that SGML is all that – and more. In the introduction to [1] Yuri Rubinsky wrote:

> ISO 8870 never describes SGML as a meta-language, but everything about its system of declarations and notations implies that a developer has the tools to build exactly what is required to indicate the internal structure of any type of information in a common tool independent manner.

Indeed, a strong point of SGML is that it can be regarded as a meta-language, a tool with which one can define the syntax of many languages, very much similar to context-free grammars. In SGML terminology these 'languages' are called document type definitions, called DTD for short. DTDs can he written for any type of information, research papers, books and music. A DTD can be used for many purposes, of which two important ones are storage and exchange of information coded according to this DTD.

The premise of this paper is that the exchange of information, if it is based on SGML, needs a single common DTD, agreed upon by all parties involved, for each class of documents that is exchanged

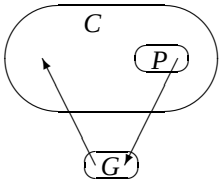
Suppose two parties, *A* and *B*, exchange information in the form of one class of documents. and that they each have a DTD, $D(A)$ and $D(B)$, with $D(A)$ not identical to $D(B)$. If *A* sends a document to *B* then *A* can include the document type definition $D(A)$. for that document (instance) at the beginning of the document. This enables *B* to use an SGML parser to check the validity of the document he received. However, there is nothing more *B* can do with the document: the DTD $D(A)$ contains no information about the meaning of the coding scheme that $D(A)$ defines, and a mapping of the document from $D(A)$ to $D(B)$ is a procedure that cannot be automated. The problem becomes even more difficult when a third party, *C*, is introduced, who accepts material from both *A* and *B*. How is *C* going to handle material with two different coding schemes?

This is where we encounter one of the weaknesses of SGML as it is being used currently, namely that it enables every party involved in this process to define and use a different DTD.

3 Scientific publishing

In the rest of this paper we concentrate on the exchange of information that occurs in scientific publishing, in particular on the exchange of papers that contain mathematical formulas and are published in research journals. Recent developments in this area formed the main reason for writing this paper. A few standards for encoding of mathematical formulas have already emerged, of which a well-known one is the AAP Standard or Electronic Manuscript Standard [2]. A DTD for mathematical formulas accompanies this standard, but it is not part of it. Another standard for mathematical formulas is the one adopted by CALS [3], and others are under development [4], [5].

The handling of mathematical formulas in scientific publishing is part of the bigger whole of information exchange within a (the) scientific community, with the publisher as intermediary, as is shown below:



The authors of research papers are the providers, *P*. The publishers are the gatherers of information, *G*. They accept information from many providers, gather this in the form of a journal issue, and distribute this. In this process, the publisher provides a quality check via the system of peer reviewing, makes notation consistent, and in some cases improves the prose. The information is distributed to a group of consumers, *C*, with the set *C* a superset of the set *P*. In this process, two sorts of information can be exchanged:

- material that is structured in the sense of being encoded according to, and checked against, some formal structural specification such as a DTD;
- material that is not structured.

At present most of the material exchanged in the process of scientific publishing is of the unstructured type. We expect that this will remain the situation in the near future. As soon as authors get the possibility of using more sophisticated tools, we expect that publishers will receive increasing numbers of papers of the structured type.

Several scientific publishers, among whom Elsevier Science Publishers, have adopted SGML as the future main tool for the process of publishing scientific articles [6], and several other publishers have made, or are expected to make, the same choice. The European Laboratory for Particle Physics (CERN), a large community of information providers, are using SGML to automate the loading of bibliographic information in their library's database [7]. For both authors and publishers it would be advantageous to agree on one DTD for the encoding of research papers. There are several reasons for this:

- Most authors do not submit all their articles to one and the same publisher every time me. At present they are confronted with 'Instructions to Authors' that differ significantly from publisher to publisher.
- A recent trend is that authors prepare their papers with text-processing software on some computer. This enables them to send the paper in electronic form (electronic manuscript or 'compuscript') to the publisher. Publishers are confronted with a variety of text-processing software on a variety of computer systems [8], [9]. Moreover, every field of science appears to have its own 'Top Ten' of most used text processing packages.
- Bibliographic information about all research papers in all (or most) scientific journals is stored in bibliographic databases. In an ideal world, authors would still be able to use their favourite text-processing system, which would generate SGML 'behind the screens', so to speak. All publishers would accept one standard DTD, and all text-processing systems would be able to generate documents prepared according to this DTD, and all bibliographic databases would be able to store this material.

An example of activities towards achieving this ideal situation: the European Working Group on SGML (EWS) and the European Physical Society (EPS) have taken the Electronic Manuscript Standard and are trying to develop it into a complete DTD, which should be acceptable to information providers, information gatherers and information consumers. The Electronic Manuscript Standard is now a Draft International Standard, ISO/DIS 12083. The EWS and EPS hope that the final standard will include their work.

4 Encoding of mathematical formulas

In Annex A of ISO 8879 [10] we find the following:

> Generalized markup is based on two novel postulates:
>
> - Markup should describe a document's structure and other attributes rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.
> - Markup should be rigorous so that the techniques available for processing rigorously defined objects like programs and databases can be used for processing documents as well.

There is no reason why this should not be valid for mathematical formulas. We need to delimit the kind of mathematical formulas we are trying to describe if we want an unambiguous structure. The field of mathematics is so vast, that it may be impossible to design a single DTD that covers every kind of mathematical formula. If we concentrate on those sciences which use mathematics as a tool, for example physics, we see that the mathematics used in many physics papers can be described as "advanced calculus" This definition can be made more precise by referring to some

standard textbooks containing these types of formulas, e.g. Handbook of Mathematical Functions [11] and the Table of integrals, series and products [12].

If we aim for rigorous encoding of mathematical formulas (the second postulate), we must develop a system of descriptive markup of mathematical formulas that enables us to:

- convert the formulas between different word processors;
- store the formulas in and extract them from a database;
- allow programs to input or output formulas in descriptive markup.

An example of the first application would be the conversion of mathematical formulas coded in LaTeX to, say, Word[1] via SGML. The benefits of using SGML as an intermediate language for conversion are described in [13]. Note, for example, that the number of programs required for pairwise conversion between n languages is proportional to $n^2 - n$ without an intermediate language, but to $2n$ with an intermediate language.

An example of the second application would be encoding and storing the complete contents of the above mentioned Handbook of Mathematical Functions [11] and Table of integrals, series and products [12] in a database, so that this information can be accessed on-line by, say, mathematicians and physicists. Many articles have mathematical formulas in their titles, so any program that extracts bibliographic data should be able to handle mathematics as well.

An example of the third application would be the extraction and subsequent use in a computer program, written in an ordinary programming language or, for example, in Mathematica.[2]

At this point we come back to the ideal world for scientific publishing we sketched earlier. In this world, publishers would use one standard DTD for scientific papers, which enables them to prepare a primary publication – in paper and (or) in some electronic form - and to store the information in databases for various secondary purposes.

The question now is: what should a DTD for mathematical formulas look like, if it is going to be used for these purposes?

There are two choices for a DTD for mathematics:

- P-type: the DTD reflects the Presentation or visual structure; examples of this type are discussed in the appendices.
- S-type: the DTD reflects the Semantics or logical structure; at present no DTDs of this type exist.

The quotation from Annex A of ISO 8879 [10] indicates the preference of the creator(s) of SGML: markup of a formula should be of S-type, it should describe the logical structure of the formula, rather than the way it is represented on a certain medium, say the page of a traditional (non-electronic) book.

Let us suppose, for the sake of the argument, that an information gatherer, a publisher, chooses a DTD of S-type. This raises two further questions:

1. Is descriptive markup of mathematical material possible?
2. If it is possible, who can use it and for which purposes?

The second question needs some explanation. As discussed in section 3Scientific publishingsection.17, in the process of scientific publishing two sorts of information can be exchanged. mathematical material that is structured according to a formal structural specification, and material that is not structured. This means that there are two possible scenarios.

Scenario 1: an author submits a paper in the form of a manuscript (paper), i.e. with unstructured formulas, or a compuscript with mathematical formulas in P-type notation (TeX, WordPerfect, ...).

Scenario 2: an author submits a paper with mathematical formulas in S-type notation. In scenario 1 it is the task of the publisher to convert from paper or P-type notation to S-type notation. Before we discuss the feasibility of this conversion, we will first look at some characteristics of mathematical notation.

4.1 Characteristics of mathematical notation

Mathematical notation is designed to create the correct ideas in the mind of the reader. It is deliberately ambiguous and incomplete: indeed, it is almost meaningless to all other readers. Or, more technically: the intrinsic information content of any mathematical formula is very low. A formula gets its meaning, i.e. its information content, only when used to communicate between two minds which share a large collection of concepts and assumptions, together with an agreed language for communicating the associated ideas.

The ambiguity encountered in mathematical notation can be of two types [14]

1. A generic notation uses the same symbols to represent similar but different functions, for example '$+$' or '$\times$'. In the case of addition this is not really a problem, but multiplication is a problem since, multiplication of numbers is commutative, whereas matrix multiplication is non-commutative!

[1] Word is a registered trademark of MicroSoft.
[2] Mathematica is a registered trademark of Wolfram Research.

2. A more fundamental ambiguity is posed by the same notation being used in different fields in different ways. For example: f' stands for the first derivative of f in calculus, but can mean 'any other entity different from f' in other areas.

More examples of ambiguity are:

- Does $\bar{x}$ represent a mean, a conjugation or a negation?
- Is i an integer variable, e.g. the index of a matrix, or is it $\sqrt{-1}$?
- The other way around: is $\sqrt{-1}$ denoted by i or by j?[3]
- What is the function of the 2 in SU_2 $\log_2 x$, x^2, T_2^2?[4]
- Is $|X|$ the absolute value of a real (complex) number X or the polyhedron of a simplicial complex X [15]?

The inverse problem, which is equally common, arises when different typographical constructs have the same mathematical meaning. For example, the meanings of both the following two lines would be coded identically

$$\begin{array}{lll} 3 & + & 4(\mathrm{mod}\,5) \\ 3 & +_5 & 4 \end{array}$$

and this would lead to great difficulty if an author wanted to write:

> We shall often write, for example, $3+4(\mathrm{mod}\,5)$ in the shorter form $3+_5 4$, or even as simply $3+4$ when this will not lead to confusion.

Of course, natural languages are similarly ambiguous and incomplete, but no one we know is suggesting that in an SGML document each word should be coded such that it reflects the full dictionary definition of the meaning which that particular use of the word is intended to have!

4.2 Who performs the markup of math?

How does one convert P-type mathematical material, which an author has produced, to S-type notation, which the publisher uses? In [1], (p.9) Goldfarb gives a three-step model for document processing:

1. recognition of part of a document (adding a generic identifier for the appropriate element);
2. mapping (associating a processing function with each element);
3. processing (e.g. translating elements into word processor commands).

In the publishing of scientific papers and books steps 2.Who performs the markup of math?Item.30 and 3.Who performs the markup of math?Item.31 are the responsibility of the publisher. Traditionally, step 1.Who performs the markup of math?Item.29 was also their responsibility: the technical editor adds markup signs in the margin of the manuscript, depending on the text and the visual representation that the house style dictates. It is, however, unlikely that a technical editor is capable of identifying the precise function of every part of a mathematical formula, for several reasons, most of which were discussed in the previous subsection, namely that mathematical notation:

- is not unambiguous,
- is not completely standardized,
- is not a closed system.

Even if the technical editor were capable of identifying every part of a formula, this would be too time- consuming – and therefore too costly. However, under certain conditions [16], automatic translation from visual structure to logical structure of mathematical material is simplified greatly.

This, and what we discussed in section 4.1Characteristics of mathematical notationsubsection.23, leads us to conclude the following. A publisher has no choice but to use a P-type DTD for mathematical material that is submitted in unstructured form or in P-type notation. Even if S-type markup of a mathematical formula would be possible, conversion from P-type to S-type would be difficult or even impossible. Conclusion: the tags for S-type markup should not be added by the information gatherer, but by the information providers, i.e. the authors, who should be able to identify each part of their formulas.

[3] There are examples of authors actually writing something like $[L_i, L_j] = \frac{i}{2} L_k$, where the first i is an index, and the second i stands for $\sqrt{-1}$.

[4] In SU_2 it is the number of dimensions of the Lie group; in $\log_2 x$ it is the base of the logarithm; if x is a vector, the $_2$ in x_2 is an index: the ^{2} in x^2 could be a power, but if T is a tensor, the ^{2} in T_2^2 is a contrainvariant tensor index.

4.3 Feasibility of S-type notation

In our second scenario, authors would submit papers with mathematical formulas in S-type notation. This would enable the publisher to 'down translate'[5] to any mathematics typesetting language (P-type notation). However, the same reasoning as in section 3.1 leads us to the following conjecture:

Conjecture. It is impossible to create an S-type DTD for all of mathematics.

Representing the "full meaning" of a mathematical formula, if such a notion exists, will almost certainly lead to attempts to pack more and more unnecessary information into the representation until it becomes useless for any purpose. This is rather like Russell and Whitehead reducing "simple arithmetic" to logic and taking several pages of symbols to represent the "true meaning of $2+2=4$".

Even if it were possible to define an S-type DTD for a certain branch of mathematics, this still gives problems. Supposing an S-type DTD contains an element for a "derivative" of a function. Since the S-type DTD will not contain any presentational attributes, a decision will have to be made to represent the derivative of $f(x)$ on paper as $f'(x)$ or $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$. There are, however, times (such as in this article) that both representations are required for the same semantic object, and that the author will need other notation in addition to that defined by the S-type DTD.

A likely reason for the belief that an S-type DTD is possible, is that many people in the worlds of document processing or computer science are convinced that each symbol has at most a few possible uses and that mathematical notation is as straightforward to analyse as, for example, a piece of code for a somewhat complicated programming language. The reality is that mathematical notation is more akin to natural language: it is ambiguous and incomplete, as we pointed out earlier.

4.4 Some problems with existing languages

To show that it is not obvious to capture mathematical syntax in a DTD, let alone its semantics, consider the example of a limit

$$\lim_{x \to a} f(x)$$

The syntactic structure of a limit is:

- The limit operator
- The part containing the variable and its limit value
- The expression of which the limit is to be taken

The first part could:

- always be "lim", in which case it is just a part of the presentation of the formula and it should be left out.
- be one of a finite list of alternatives, indicating the type of limit(lim inf, sup, max, etc.). In this case it should be an attribute.
- be any expression.
- be any text.

We think the second possibility comes closest to the syntax of the limit construct. Th second and third parts can be any mathematical expression.

Now let's look at the way this formula is coded with the DTDs from ISO TR 9573, AAP math and Euromath respectively. Using the mathematics DTD from ISO TR 9573 there are three possibilities:

- `lim <sub pos=mid> x &rarr; a </sub> f(x)`
- `<plex><operator>lim</operator><from>x &darr; a</from> <of>f(x)</of></plex>`
- `<mfn name=lim><sub pos=mid>x &rarr; a</ll><opd>f(x)</opd></lim>`

whereas with the Euromath DTD we would have:

```
<lim.cst><l.part.c limitop=lim><range>
<relation>x\&rarr; a </relation></range>
</l.part.c><r.part.c><textual>f(x)</textual>
```

We see that the AAP and Euromath expressions are closest to the limit syntax. The best solution from ISO TR 9573 involves a more general "plex" construct, which can be used for integrals, sums, products, set unions, limits and others. When the plex construct contains the actual lower and upper bounds it may even give semantic information.

Some mathematicians, however, are not satisfied with this solution [17]. The plex operation is probably a notation for an iterated application of a binary operation (e.g. sums and products), while limits are of a different nature. In many

[5] 'Down' because information is lost in the process; we borrowed the terminology of translating 'up' and 'down' from Exoterica OmniMark.

cases only the from part will be used, and there the whole range of the bound variable will be indicated, as an interval or a more general set. How does one go about extracting the bound variable?

This supports our conjecture from the previous section, namely that it is very hard to capture the semantics for all mathematics. it also suggests that some redundancy is required to select whichever notation is most appropriate in a certain context.

5 Re-using mathematical formulas

There are two important uses for a generically coded mathematical formula. The first one is in a mathematical manipulation – or computer algebra – system (MMS), such as Mathematica [18] or Maple [19]. Computer programs for the numerical evaluation of formulas, for example written in FORTRAN or Modula-2, can also be regarded as mathematical manipulation programs.

The second form of re-usage is in a mathematical typesetting system, for formatting the formula on paper or on screen; examples of this are TEX [20] and eqn/troff [21], [22].

For computer algebra systems the notation for the formula should be such that a particular type of manipulation on a particular system is possible, given a 'background' of concepts and assumptions that enables the system to interpret the input as a mathematical statement.

The coding of a formula that is adequate for document formatting, for example the TEX notation `f^{(2)}(x)`, is very unlikely to contain much of the information required for a manipulation system to make use of it. However, for a limited held of discourse it is feasible to use the same coding for both types of system [16].

Some examples: the square of $\sin x$ is typographically represented as $\sin^2 x$, but a system like Mathematics or Maple would probably prefer something like $(\sin x)^2$ as input. Typesetting the inverse of $\sin x$ as $\sin^{-1} x$, however, could be confusing: does it mean $1/(\sin x)$ or $\arcsin x$?

An MMS would probably require the second derivative of a function f with respect to its argument x to be coded as $(D,x)((D,x)f(x))$ but on paper this would be represented as $f''(x)$, or $f^{(2)}(x)$, or $\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}$.

On the output side of a MMS there are other problems since some of the coding necessary for typographically acceptable output cannot be automatically derived by the system from the coding used by the MMS.

The Euromath view [17] is that a common interface should be designed together with the manufacturer of a MMS. Perhaps an MMS-type DTD will be required.

6 Related problems

Another problem is, of course, that mathematics is by its nature extensible, so there will always be new types of manipulations to be done. Notations are changed or new notations are invented almost every day, figuratively speaking. Normally these new subjects will use existing typographic representations, but the computer algebra system will not know what formatting to use! Occasionally a new typographic convention will be needed. And although there is agreement on the notation for most mathematical concepts, authors of books on mathematics tend to introduce alternative notations, for instance when they feel this is necessary for didactic reasons. Mathematical notation is not standardized, and it is open – anyone can use it, and add to it, in any way they wish.

If we consider a given DTD at any time, we have to ask ourselves: can an author add elements when the need for this arises? Theoretically the answer is 'Yes, he can' [23], (p.71), although it is not straightforward to include the new elements in the content models of existing elements.

Are such modification by the author desirable? A DTD which is locally modified by an author will quickly give rise to the situation described in the introduction to this paper, and this should therefore probably be discouraged. Others, however, have also noticed a need for private elements, as described in EPSIG News 3, no. 4; one of the challenging aspects of using SGML being encountered by the Text Encoding Initiative is that the guidelines need to be extensible by researchers. They need to be able to extend the DTD in a disciplined way.

This problem, however, may not be a serious one. The collection of style elements is almost a closed set, since the number of fonts, symbols and ways to combine them is limited. In fact, most notation is not syntactically new, since the limited number of constructs works well as a notation. The multitude of notations is obtained by combinations of fonts, symbols and positions (left or right subscript, left or right superscript, atop, below, ...), and by giving one notation more than one meaning. This again seems to support our view that only a P-type DTD can be constructed for *all* of mathematics.

An SGML DTD, of whatever type, also doesn't solve the problems of new atomic or composite symbols, which

occur frequently in mathematics. As with new elements, an author can add entities for these new symbols. There is no method to add the name of a new symbol, whether atomic or composite, to an existing set of entity definitions for symbols, other than to contact the owner of the set and wait for an update.

Although there is now a standard method to describe that symbol's glyph (shape) [24], it is not practical for an author to include it. A compromise solution seems to be to extend an existing set, such as the one from ISO [25], as much as possible, and try to standardize its use.

7 Conclusions

We have argued as follows:

- That a logical DTD in the sense of describing the structure of the mathematical meaning is as impossible for maths as it is for natural language, and also it is useless for formatting since the same mathematical structure can be visually represented in many different ways. The correct one for any given occurrence of that structure cannot be determined automatically, but must be specified by the author.
- That what needs to be encoded for formatting purposes, is information that enables a particular set of detailed rules for maths typesetting to be applied. This could he described as a 'generic-visual encoding' or 'encoding the logic of the visual structure'. To establish exactly what these code?, should be will require an expert analysis (probably involving expertise from mathematicians, particularly editors, and from typographers aware of the traditions of mathematical typesetting).
- That this is different to what needs to be encoded for use in mathematical manipulation software. Since neither of these encodings can be deduced automatically from the other, a useful database will need to store both. Perhaps a separate DTD will be required to enable this communication.

Possible solutions are

- A DTD based on a hybrid of visual structure and logical structure
- Two DTDs, one for visual structure and one for logical structure, that are linked in some fashion
- Two concurrent DTDs, one for visual structure and one for logical structure.

The simplest solution is probably to have a basic visual structure which is described as an SGML entity, supplemented with a (redundant) logical structure, described by a second SGML entity. This solution avoids any special SGML features and gives the user all flexibility for mixing and matching as required. We believe that similar reasoning can be applied to tables and chemical formulas, where the problem of separation form from content is just as complex, or even more.

References

[1] Charles Goldfarb. *The SGML Handbook*. Oxford University Press, Oxford, 1990.

[2] Standard for electronic manuscript preparation and markup version 2.0. Technical Report Z39.59-1988, ANSI/NISO, 1987.

[3] Techniques for using SGML. Technical Report 9573, ISO, 1988.

[4] American Chemical Society. ACS journal DTD.

[5] Björn von Sydow. On the `math` type in Euromath.

[6] N. A. F. M. Poppelier. SGML and TEX in scientific publishing. TUGboat, 12:105–109, 1991.

[7] E. van Herwijnen, N. A. F. M. Poppelier, and J.C. Sens. Using the electronic manuscript standard for document conversion. *EPSIG News*, 1(14), 1992.

[8] E. van Herwijnen. The use of text interchange standards for submitting physics articles to journals. *Comp. Phys. Comm.*, 57:244–250, 1989.

[9] E. van Herwijnen and J.C. Sens. Streamlining publishing procedures. *Europhysics News*, pages 171–174, November 1989.

[10] Standard generalized markup language (SGML). Technical Report 8879, ISO, l986.

[11] M. Abramovitz and I. Stegun. *Handbook of mathematical functions*. Dover, New York, 1972.

[12] I.S. Gradshteyn and I.M. Ryzhik. *Tables of integrals, series, and products*. Academic Press, New York, 1980.

[13] S.A. Mamrak, C.S. O'Connell, and J. Barnes. Technical documentation for the integrated chameleon architecture. Technical report, March 1992.

[14] Neil M. Soiffer. *The design of a user interface for computer algebra systems*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, 1991. Report UCB/USD 91/626.

[15] M. Nakahara. *Geometry, Topology and Physics*. Adam Hilger, Bristol, 1990.

[16] Dennis S. Arnon and Sandra A. Mamra. On the logical structure of mathematical notation. TUGboat, 12:479–484, 1991.

[17] Björn von Sydow. private communication to EvH.

[18] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley, Reading, 1991.

[19] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, and Stephen M. Watt. *Maple User's Guide*. WATCOM Publications Ltd., Waterloo, 1985.

[20] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, 1984.

[21] Joseph E Osanna. Nroff/troff. In *UNIX Programmer's Manual (2b)*. Bell Laboratories, 1978.

[22] Brian W. Kernighan and Linda Cherry. Typesetting mathematics. In *UNIX Programmer's Manual (2b)*. Bell Laboratories, 1978.

[23] E. van Herwijnen. *Practical SGML*. Kluwer Academic Publishers, Dordrecht, 1990.

[24] Font information interchange. Technical Report 9541, ISO, 1991.

[25] Information processing – SGML support facilities – techniques for using SGML – part 13. Technical Report 9573, ISO, 1991. Proposed Draft Technical Report.

III A LaTeX Tour, part 3: mfnfss, psnfss and babel

David Carlisle
david@dcarlisle.demon.co.uk

1 Introduction

This third installment of my tour covers three more distributions that are supported via the standard LaTeX bug report mechanism described in Part 1.

The mfnfss distribution provides LaTeX support for some popular Metafont produced fonts, that do not otherwise have any LaTeX interface.

The psnfss distribution consists of LaTeX packages giving access to PostScript fonts.

The third distribution in this part of the tour is babel, which provides LaTeX with multi-lingual capabilities.

2 The MFNFSS Distribution

The mfnfss distribution is something of a 'collecting point' for files in the distribution that have not got anywhere else to go.

2.1 Font Packages

These packages provide LaTeX interfaces to some publicly available fonts. They do *not* provide the fonts themselves, which are available from the `fonts` tree in the standard CTAN archives.

`pandora` The 'Pandora' family of fonts designed by Nazneen N. Billawala is an alternative to the standard 'Computer Modern' fonts of Knuth. The family consists of a full range of text fonts, including sans-serif and slanted.

`oldgerm` The old German fonts designed by Yannis Haralambous. There are three styles of text font, Schwabacher, Fraktur and Gothic. (The terms 'Fraktur' and 'Gothic' tend to be used interchangeably by English speaking mathematicians such as the present author, but the fonts in this collection have clearly distinguishable styles.)

There is also a font of 'initials', highly ornate uppercase letters, suitable for use as the first letter of a section. If you wish to use this in 'drop caps' style you may also want to use one of the contributed packages available on CTAN such as drop, or dropping, that automate the setting of a suitable paragraph shape and inserting the initial letter at the correct size.[6]

2.2 T1 Encoded 'Concrete' Fonts

Note: The following two files require the old release 1.1 of the dc fonts. Walter Schmidt very recently (March 1997) released a test version of a set of 'Concrete' fonts based on the new ec font base. The LaTeX support for these new fonts is available from `macros/latex/contrib/supported/ccfonts`. Once this release is stable, the following files will probably be removed from the mfnfss distribution.

`dccr.mf` Metafont source file used by the output files from `dccrstd.tex` to generate Concrete Roman fonts in T1 encoding.

`dccrstd.tex` TeX file used in the generation of Concrete Roman fonts in T1 encoding. It will produce a number of `.mf` files corresponding to Concrete Roman fonts in different sizes. By modifying the table inside this file further Metafont driver files can be generated. The `.fd` files for the Concrete Roman fonts can be produced with `cmextra.ins` which is part of the LaTeX base distribution.

3 The PSNFSS Distribution

With the release of LaTeX 2ε, LaTeX gained inbuilt support for the use of alternative font families in documents, and in particular for the use of scalable font formats such as Type 1 (PostScript) or TrueType.

The collection of packages, coordinated by Sebastian Rahtz, known as psnfss offers convenient interfaces to most of the more common font sets.

[6] The `fd` files provided here load the original `yinit` font. The CTAN archives also contain '`yinitas`', a modified version of this font.

Most of the files here relate to font files renamed to a consistent naming scheme, promoted and maintained by Karl Berry. This encodes the font vendor, and details of the font such as its weight, style and encoding into a compact name that usually fits in the eight letter filenames used by some common filesystems. More information about the font naming scheme can be found on CTAN in `info/fontname`. It should be noted however that the packages themselves, such as the times package, do *not* depend on any particular font naming convention. LaTeX isolates packages from the details of the external font files by the use of '`fd`' (Font Descriptor) files which map the LaTeX 'NFSS' model of fonts to the external font metric files.

In principle, there is no real need for packages to load text fonts into LaTeX. For example, once the font metrics and font descriptor files for Times Roman (which is ptm in the Karl Berry Naming Scheme) are installed, then one could in principle switch to Times Roman in a LaTeX document by simply specifying `\fontfamily{ptm}\selectfont`. Normally one would instead want to assign the new font to one of the 'default' LaTeX families, Roman, as used by `\rmfamily`, Sans Serif (`\sffamily`) and Typewriter or Monospace (`\ttfamily`).

The support for PostScript fonts is split into two. The CTAN `fonts/psfonts` area contains material that is mainly automatically generated from the Adobe font metric files that are distributed with all Type 1 fonts. This includes the font metrics themselves, the Font Descriptor files, the 'map' files used to make fonts known to the dvips driver, and some basic packages to declare single fonts to LaTeX. This is supplemented in `macros/latex/packages/psnfss` by the 'hand written' packages of the psnfss collection that load popular *combinations* of font families, or deal with mathematics.

This section refers at various points to PostScript or Type 1 fonts, but in fact the TeX support for these fonts applies equally well to True Type, or other scalable formats. As long as TeX has access to the font metrics, the font format does not matter (to TeX; it matters to the driver you use to print the DVI file).

3.1 PSFONTS

The CTAN psfonts area primarily contains the font metric and LaTeX font descriptor files, organised by font vendor, as outlined below. The basic format of the file structure is the same for each font family, so only the top level directories are given here, except for the Adobe Times family, which is further expanded as an example.

Font Vendors

The font subdirectories of `fonts/psfonts` are:

`adobe` Fonts sold by Adobe, or built into PostScript devices.
`bh` Fonts designed by Bigelow and Holmes, these are mainly sold through Y&Y.
`bitstrea` Bitstream fonts.
`monotype` Monotype fonts.
`textures` Textures Fonts for the Blue Sky Research Macintosh TeX implementation.
`urw` Fonts distributed by URW.
`xadobe` Adobe 'expert' font sets.
`xmonotype` Monotype 'expert' font sets.

Each of the vendor directories contains subdirectories corresponding to the font families supported by the psfonts distribution. (Using the tools provided one can generate TeX support files for most other text fonts, the selection here is really just a set of examples.)

The subdirectories of the `adobe` directory are:

`agaramon` Adobe's rendition of a Garamond serif Roman family. (Commercial.)
`avantgar` Avant Garde sans serif (built into most PostScript devices).
`baskervi` Baskerville, a commercially available serifed Roman family. (If you are reading this in *Baskerville* then it is similar to the text you see, which is Monotype Baskerville).
`bembo` Bembo, a commercially available serifed Roman family.
`bookman` Bookman (built into most PostScript devices).
`centaur` Centaur, a commercially available serifed Roman family.
`courier` Courier (built into all PostScript devices).
`garamond` Garamond 3. Another Garamond serif Roman family. (Commercial.)
`gillsans` Gill Sans, a commercially available sans serif family.
`helvetic` Helvetica (built into all PostScript devices).
`nbaskerv` ITC New Baskerville, another variant on the Baskerville theme. (Commercial.)
`ncntrsbk` New Century Schoolbook (built into most PostScript devices).
`optima` Optima, a commercially available sans serif family.

`palatino` Palatino serifed Roman family (built into most POSTSCRIPT devices).
`symbol` Symbol (built into all POSTSCRIPT devices).
`times` Times Roman (built into all POSTSCRIPT devices).
`univers` Univers, a commercially available sans serif family.
`utopia` Utopia, a commercially available serifed Roman family.
`zapfchan` ITC Zapf Chancery. A script font built into most POSTSCRIPT devices.
`zapfding` ITC Zapf Dingbats. A symbol font built into most POSTSCRIPT devices.

All the directories corresponding to a font family look essentially the same, each with the following subdirectories.

`dvips` Contains the 'map' file for the dvips driver program. This file can be appended to `psfonts.map` or used via a configuration file to tell dvips where to find the specified fonts. A suitable configuration file is included in the directory.

Other drivers will need similar information, but perhaps in a different format.

`tex` This directory contains the font descriptor files which must be placed in the input path for LaTeX, so that LaTeX has available the information about the available fonts. For some font families this directory would also contain a LaTeX package that assigns the fonts to one of the standard LaTeX font families, such as `\sffamily`. Some packages, such as times, are not distributed here as they would clash with the packages distributed as part of psnfss, as described below.

`tfm` The font metrics, in 'tfm' format. These files contain all the information about letter sizes, ligatures, and kerning that TeX needs to typeset text.

There are several files, as each font in the original family is made available in several encodings, the two main ones being the 'Classic' TeX encoding used by Computer Modern. This is known as OT1 in LaTeX, and as '7t' in the Karl Berry font naming scheme used here. Similarly the files with names ending in '8t' relate to fonts encoded to the eight bit 'Cork' encoding, known as T1 in LaTeX.

`vf` The virtual fonts. Most (but not all) drivers handle the re-encoding of the original fonts to the encodings that TeX expects by means of the virtual font mechanism. Some special fonts, such as Zapf Dingbats are not re-encoded, and so do not have a `vf` directory.

There is one very important thing to note about the above list. *There are no fonts!* Almost all of the `fonts/psfonts` area of CTAN is concerned with providing mechanisms for using fonts that you have obtained *elsewhere*. The fonts may be built in to your printer, or may be purchased separately. There are a few freely available Type 1 fonts. In such cases there will be an additional directory, `type1`, which contains the font files (normally in 'pfb' format).

Standard POSTSCRIPT *Fonts*

In addition to the above directories, the psfonts area contains two zip files. If you need the files and have not got unzip (or pkunzip or winzip or...) then you can get a copy of unzip from the CTAN support area.

`lw35nfss` This zip archive expands to the subset of the `psfonts/adobe` tree that corresponds to the 'Standard 35' POSTSCRIPT fonts as used in Adobe Laserwriter printers. If you are only interested in using fonts built into your printer, and not in using downloaded fonts, then just get this file rather than the large collection of metrics in `psfonts/adobe`.

`lw35pk` This zip archive contains bitmap fonts for the 'Standard POSTSCRIPT fonts' in the usual PK format understood by most dvi drivers. This enables documents using Type 1 fonts to be previewed with dvi previewers that can not use outline font formats. (For example xdvi or the emtex drivers).

Tools and Extra Packages

There are a few remaining directories in psfonts.

`ts1` The LaTeX textcomp package and related utilities for accessing fonts in the 'text companion' encoding known as TS1 in LaTeX. These include the TC fonts that are distributed with the EC fonts, and suitably re-encoded fonts from the standard Type 1 font sets. This encoding contains many non alphabetic symbols that should match the current text font (rather than the math font). It includes currency symbols, superior digits, dagger signs, etc.

`mathcomp` A contributed package for using the text companion fonts in math mode.

`tools` The source for the scripts and utilities used for generating all these files.

3.2 Standard PSNFSS Packages

By contrast to the packages and font descriptor files in the psfonts distribution, the psnfss distribution contains 'hand written' files. These are either used to set up popular *combinations* of the 'standard' fonts, or load alternative font sets for mathematics. Due to the nature of mathematics fonts, these latter packages are typically much more complicated

internally than the one or two line packages that load text fonts. For the user, however, this complexity should not be apparent.

The first set of packages (all generated from the source file `psfonts.dtx`) load combinations of the Basic Adobe PostScript font set into LaTeX.

`times` As one might guess, this declares Times Roman as `\rmfamily`. For mainly historical reasons, this package also declares Helvetica as `\sffamily` and Courier as `\ttfamily`, so effectively ensuring that all text (but not mathematics) is set in the basic PostScript font set.

This is a convenience for the user who wants to replace all the text fonts by references to the basic Adobe fonts. It is an advantage to do this if you want to produce device independent and small PostScript documents for distribution. The disadvantage is that Times Roman, Helvetica and Courier, despite being the 'standard PostScript combination' look particularly horrible if placed next to each other at the same nominal size, as done by this package. Helvetica has a much larger 'x-height' (the height of the lower case letters) than Times Roman, so if sans serif and Roman text are mixed in-line, then the sans serif looks much too big. (This is not so much of a problem if the sans serif is only used for headings.) Courier is just too 'wide' when placed alongside Times Roman, which is a particularly compact font.

To partially compensate for these problems, the pslatex package (written by me, but currently distributed as a contributed package, not part of the core LaTeX distribution) is an alternative to the times package. It loads Helvetica scaled by 90% and loads Courier by way of a virtual font that condenses it by scaling the horizontal direction (only) by 85%. pslatex also contains a copy of the mathptm package (see below) so installs a Times-Italic based font set for use in mathematics.

`palatino` Declares Palatino as `\rmfamily`, and Helvetica and Courier as `\sffamily` and `\ttfamily`.

`helvet` Declares Helvetica as `\sffamily`. (Does not change the other families.)

`avant` Declares Avant Garde as `\sffamily`. (Does not change the other families.)

`newcent` Declares New Century Schoolbook as `\rmfamily`, Avant Garde as `\sffamily` and Courier as `\ttfamily`.

`bookman` Declares Bookman Roman as `\rmfamily`, Avant Garde as `\sffamily` and Courier as `\ttfamily`.

`chancery` Declares Zapf Chancery as `\rmfamily`.

The above packages only affect *text* fonts, not mathematics. `psfonts.dtx` contains one special package, written by Alan Jeffrey, which does affect the math setup.

`mathptm` This package uses a set of virtual files that use various built in or freely available fonts to make a set of fonts suitable for replacing the standard Computer Modern Math fonts. In the current release, bold fonts (and so the LaTeX `\boldmath` command) are not supported. The pslatex package referred to above contains an essentially verbatim copy of mathptm.

One may use mathptm as an example of the coding needed to make virtual fonts for mathematics based on other text italic fonts. How successful this will be depends to a certain extent how visually compatible are the symbols that are gathered from the various 'real' fonts that are used by the virtual math fonts. There are often good reasons for making such fonts (the main one being that documents using freely available fonts may be more easily placed on the Web in PostScript form), however the result is never likely to be as good as using fonts that have symbols that are *designed* to be visually compatible. For mathematics use within TeX, that currently restricts use to Computer Modern, or the commercial MathTime or Lucida Bright font sets described below.

The `psfonts.dtx` source file contains one other package:

`pifont` This declares the Zapf Dingbats font which contains an assorted mixture of symbols, and also defines new user level commands to access these symbols. See the package documentation, or *The LaTeX Companion* for details.

3.3 Freely Available Type 1 Text Fonts

The next set of packages are contributed by Peter Dyballa. In fact these are just one-line packages loading the appropriate font. Most of the code is in the `fd` files which are generated from the same source file.

`charter` Defines `\rmfamily` to use Bitstream Charter.

`nimbus` Declares URW Nimbus Roman-Regular and URW Nimbus Sans-Regular as `\rmfamily` and `\sffamily`. These are essentially free clones of Times Roman and Helvetica.

`utopia` Defines `\rmfamily` to use Adobe Utopia-Regular.

3.4 Commercial Text Fonts

The following packages are generated from the source file `adobe.dtx`. They are a rather random selection from the large catalogue of fonts sold by Adobe.

`garamond` Garamond as `\rmfamily`, Optima as `\sffamily` and Courier as `\ttfamily`.
`basker` Baskerville as `\rmfamily`.
`mtimes` Monotype[7] Times as `\rmfamily`.
`bembo` Bembo as `\rmfamily`, Optima as `\sffamily` and the ever popular Courier as `\ttfamily`.

3.5 Adobe Lucida

The following two packages relate to the original Lucida font set, designed by Bigelow and Holmes and sold by Adobe. They are generated from the `alucida.dtx` source file.

`lucid` Declares Lucida Roman and Lucida Sans as the Roman and sans serif families, and Adobe Courier again as the monospaced font.

`lucmath` Lucida has a matching set of mathematics fonts suitable for TEX use. This package makes the required definitions to make these known to LATEX.

3.6 Lucida Bright

A newer and more extensive Lucida family, also designed by Bigelow and Holmes but in this case sold by Y&Y, is known as 'Lucida Bright' and 'Lucida New Math'. The LATEX support described here was written by Sebastian Rahtz and myself.

`lucidabr.dtx` This package (replacing the earlier lucidbrb and lucidbry packages) changes the LATEX defaults for both text and mathematics to use the Lucida Bright and Lucida New Math font collections. It has numerous options to control different aspects of the package and to control which of the fonts to use. (Lucida Bright contains several font families, including 'fax' and 'casual' etc, as well as variant forms of the math italic alphabet.)
The LATEX package and the font descriptor files for the math fonts are generated from this source file. The font descriptor files for the Lucida text fonts in the standard LATEX encodings are available from the psfonts area (in the bh) directory, after Bigelow and Holmes, the creators of these fonts.
The TEX support and font metrics are freely available, but the fonts themselves must be purchased separately.

`lucidabr.ins` LATEX installation file for Lucida Bright using the standardised 'Karl Berry' font names.

`lucidabr.yy` Alternative installation file. Use this instead of `lucidabr.ins` if you plan to install the fonts with their original font names, as sold by Y&Y. (In this case you do *not* need the `fd` files from the psfonts area.)

`lucidabr.txt` Introduction and installation guide for this package.

3.7 MathTime

The MathTime fonts are produced by Michael Spivak 'TEXplorators'. They are sold by Y&Y. The LATEX support was written by Frank Mittelbach and myself.

`mathtime.dtx` The mathtime package is mainly concerned with mathematics setup, although it selects Times, Helvetica and Courier as the text fonts if they have not already been set by another package. The MathTime mathematics fonts are specially designed to match Times Roman, but blend quite well with other text fonts that are of a similar weight. Computer Modern mathematics tends to look very 'light' if used with font families other than Computer Modern. The package has several options to control the font choices made.

`mtfonts.fdd` The source for the font descriptor files for MathTime mathematics fonts.

`mathtime.ins` Installation file. Note that this file may be edited in a couple of places depending on whether or not you have the extended 'MathTime Plus' font set which includes bold math support.

`mathtime.txt` Introduction and installation guide for this package.

3.8 Documentation and Other Files

`readme.txt` General introduction.
`psnfss2e.tex` User level documentation on the use of these packages.
`test0.tex` Testing accents and other encoding specific commands are working correctly using POSTSCRIPT fonts.
`test1.tex` Test document that uses most of the 'Standard 35' fonts.
`pitest.tex` Test of the pifont package.
`mathtest.tex` Test of the mathptm package.
`makefile` Unix 'make' utility to automate installation of the packages.
`allpspk` Unix script that makes a test document using a specified font family and then uses dvips and its associated scripts to generate 'pk' versions of the fonts.
`makepk` Unix script that calls allpspk on some common fonts.

[7] Not sure why this is generated from *adobe* source file.

3.9 PSNFSSX

Recently the psnfss collection has aquired a close cousin, psnfssx, distributed as a contributed package from `macros/latex/contrib/supported/psnfssx`. This contains some lesser used or nonstandard packages, related to POSTSCRIPT support. Of particular interest might be the `ly1` files (contributed by myself) in that directory which provide the LaTeX support for the 'texnansi' encoding promoted by Y&Y by way of an `LY1` option to the fontinst package.

This psnfssx collection also contains some obsolete versions of packages formerly in psnfss; this material is provided for historical interest only. Use at own risk!

4 The Babel Distribution

The babel package is distributed from `latex/packages/babel` and is supported via the LaTeX bug reporting address, but has origins predating the current LaTeX release. As well as supporting LaTeX it contains support for plain TeX (and formats such as AMSTeX or eplain that are based on plain). Primarily babel is the work of Johannes Braams, with contributions for specific language files by numerous people.

Babel consists of a 'kernel' that extends LaTeX with a mechanism for switching between specified languages. Part of this kernel (related to hyphenation) must be loaded when the LaTeX format is made to get the full benefit of hyphenation tables for multiple languages. For each language, or related group of languages, supported by babel there exists a language-specific code file. This will offer translations of the fixed text strings used in the standard LaTeX classes, such as 'Table of Contents', 'Figure', etc., and may also offer language-specific 'shorthands' that make typing common constructs easier (for example the german option provides the construct '`"ff`' to produce 'ff' that would hyphenate to 'ff-f' if it fell at the end of a line). The language file may also modify the typesetting to support the normal conventions of that language. For example the french option modifies the spacing around punctuation marks in text.

4.1 Babel Kernel

`babel.sty` The main interface to babel. The user specifies all languages to be used in a document as options to this package, the last option specified is the default language for the document. So for example

```
\usepackage[french,german]{babel}
```

would enable the use of French and German conventions within the document, with the default language being German.

`hyphen.cfg` The standard LaTeX interface to hyphenation. When the LaTeX format is being made, this file is input if it exists, to setup the required hyphenation patterns. In the base LaTeX distribution there is no such file, and so a default action is taken which loads the original TeX patterns for American English. The babel distribution provides this configuration file (generated from `babel.dtx`) which defines some core functionality, and then reads `language.dat` to specify which hyphenation files to load.

`language.dat` This file must be edited to specify which language hyphenation files to load, and the name of the external file which contains the hyphenation table for each such language (and optionally a second external file, typically containing hypenation exceptions). Note that hyphenation files *must* be specified here, and so loaded when the format is made. This is a restriction of the underlying TeX system. Documents using other languages not specified here may still be processed, and babel will translate any fixed text strings, but it will not be able to correctly hyphenate that language. A default hyphenation will be used (most likely English) which may or may not be suitable depending how far the language differs from English.

`switch.def` This file is also generated from the same `babel.dtx` source. If babel is used as a package but was not used when the format was made, then the core functionality normally provided by `hyphen.cfg` will not be present. The package will detect this, and so input this file to provide the necessary definitions.

4.2 Language-Specific Files

The implementation of the language-specific code for each language within babel is contained in files with extension '`.ldf`' (language definition files). These are not directly input by the user, but specified as options to the babel package. Normally the option name is the same as the file name, except where noted below. Some similar languages or dialects are supported by the same external file, and some options are available in more than one name; such aliases are noted in parentheses in the list below.

Most languages also have a file with extension `.sty`; however this is just offered for compatibility with older versions of Babel and of LaTeX, or for use with plain TeX based formats. In normal LaTeX usage only the `.ldf` file is used.

`basque` Support for the Basque language.[8]
`breton` Support for the Breton language.
`catalan` Support for the Catalan language.
`croatian` Support for the Croatian language.
`czech` Support for the Czech language.
`danish` Support for the Danish language.
`dutch` The dutch and afrikaans options.
`english` The american (USenglish) and british (UKenglish) options. The option english refers to either British or American English, depending on the local installation.
`esperant` The esperanto option.
`estonian` Support for the Estonian language.
`finnish` Support for the Finnish language.
`frenchb` Support for the French language (the corresponding options are french (frenchb) or francais. If the french option is used then `french.ldf` will be used (from the GUTenburg french package) if it is available.
`galician` Support for the Galician language.
`germanb` The austrian and german (germanb) options.
`kannada` Support for the Indian language, Kannada.[8]
`irish` Support for the Irish Gaelic language.
`italian` Support for the Italian language.
`lsorbian` The lowersorbian option.
`magyar` The magyar (hungarian) options.
`norsk` Support for the Norwegian languages with options norsk, nynorsk.
`polish` Support for the Polish language.
`portuges` The brazil (brazilian) and portuges (portuguese) options.
`romanian` Support for the Romanian language.
`sanskrit` Support for the Sanskrit language, transliterated to latin script.[8]
`scottish` Support for the Scottish Gaelic language.
`slovak` Support for the Slovakian language.
`slovene` Support for the Slovenian language.
`spanish` Support for the Spanish language.
`swedish` Support for the Swedish language.
`turkish` Support for the Turkish language.
`usorbian` The uppersorbian option.
`welsh` Support for the Welsh language

Babel version 3.6 sees the welcome (re)introduction of support for non-latin scripts. It is probably fair to say that this support is still more experimental than the support for latin scripts. One problem, not directly under babel 'control', is that the TEX encodings for Greek and Cyrillic (corresponding to T1 for European Latin scripts) have not yet been finalised or agreed. Currently babel uses two 'locally defined' encodings, LWN and LGR.

`greek` The greek option, which utilises the 'kd' Greek fonts.
`russianb` The russian option, which utilises the 'LH' fonts.
`ukranian` Support for the Ukranian language.[8]

Two separate packages are currently in preparation which will be distributed, together with suitable fonts and hypenation tables, from CTAN. These will extend babel with options for the Ethiopian and Ukrainian languages.

4.3 Compatibility Files

The distribution contains the following two source files which generate files which enable the use of babel with formats based on plain TEX (and also the old LATEX 2.09 release).

`bbcompat` The source for compatibility mode files. Most languages are provided with a 'package' with extension `.sty`. This just inputs the corresponding language definition file and should never be needed using the normal LATEX interface.
`bbplain` The source for the `plain.def` file allowing the use of babel with plain TEX.

[8] Not in the current release, planned for babel 3.7.

4.4 Installation Script and Font Descriptor Files

`babel.ins` Unpacks the babel distribution from the documented source files
`cyrillic.fdd` Font descriptor files for Cyrillic fonts in 'LCY' encoding.
`greek.fdd` Font descriptor files for Greek fonts in 'LGR' encoding.

4.5 Documentation

ASCII Text Files

`00readme.txt` The distribution guide.
`install.txt` How to install Babel.
`install.mac` How to install Babel with OzTeX.
`CyrillicFonts.txt` Further notes on the Cyrillic installation.
`GreekFonts.txt` Further notes on the Greek installation.

TeX Documents

`tb1202` The source of the original article that appeared in *TUGboat*, Volume 12 (1991), No. 2.
`tb1401` The source of an update article that appeared in in *TUGboat*, Volume 14 (1993), No. 1.
`tb1604` The source of an update article that never appeared in *TUGboat*, but was presented at EuroTeX 1995, Arnhem.

4.6 Example File

`language.skeleton` An example file that can be used to build new language definition files from scratch.

5 Coming Soon

Part 4 of this tour will describe the files of the amsfonts and amslatex distributions of packages produced by the American Mathematical society.

IV A tutorial on using MetaPost's graph package

Sebastian Rahtz
7 Stratfield Road
Oxford OX2 7BG
UK
s.rahtz@elsevier.co.uk

1 Introduction

MetaPost is a sibling program to METAFONT, which replaces the bitmap output of the latter with PostScript and is designed more as a general-purpose drawing language than a font creation package. Although it has been around for five years or so (it has been Don Knuth's tool of choice for drawing for some time), it has only recently started becoming generally available for most users. With the release of Web2c version 7.0, MetaPost is integrated into the standard Unix and Windows 32 TEX distribution, and it is also part of the CMacTeX and OzTeX packages for the Macintosh.

Although many people find general-purpose drawing languages quite forbidding and counter-intuitive, creating nice graphs from simple data files is a common task, and the purpose of this short tutorial[9] about MetaPost is to describe its graphing support. The high-level library of MetaPost macros to draw graphs was written by MetaPost's author, John Hobby, to provide a sophisticated interface comparable to grap (see Bentleyand Kernighan, 1984). It is hoped that by giving examples of its use, more people can be encouraged to try it and (who knows?) start to explore more of MetaPost for other sorts of drawing.

MetaPost is well documented in Hobby (1992), and the graph package is described in Hobby (1993); both these documents normally form part of a MetaPost distribution.

2 Getting started

To start, a quick recipe for writing a MetaPost input file. Unlike TEX, there are no backslashes or curly braces, and commands normally end with semicolons; at the start of your file, you need to load the `graph` package with an `input` command, and the file is completed with `end;`. In between you can have one or more drawings inside `beginfig`...`endfig;`, where `beginfig` has a parameter (in round brackets) of a number which will be the suffix of the output PostScript file. A graph comes inside `draw begingraph` ... `endgraph;`, where `begingraph` has a parameter of two dimensions which set the width and height of the graph. MetaPost takes care of scaling all the drawing to fit in this area. Thus a complete MetaPost file might look like this:

```
input graph
beginfig(1)
draw begingraph(2.5in,1.75in);
gdraw "yearm.dat";
endgraph;
endfig;
end;
```

If we save this as `test.mp`, and run it with the command `mpost test.mp`, the output (under Unix) looks something like this:

```
darkstart:~/# mpost test.mp
This is MetaPost, Version 0.632 (Web2c 7.0)
(test.mp (/cdrom/share/texmf/metapost/base/graph.mp
(/cdrom/share/texmf/metapost/base/marith.mp
(/cdrom/share/texmf/metapost/base/string.mp))
(/cdrom/share/texmf/metapost/base/format.mp
```

[9] This material is taken from chapter 3 of *The LaTeX Graphics Companion*, by Michel Goossens, Sebastian Rahtz and Frank Mittelbach, published by Addison-Wesley in March 1997. Reprinted by permission of Addison-Wesley.

```
(/cdrom/share/texmf/metapost/base/string.mp)
(/root/tds/metapost/latexpp/texnum.mp))) [1] )
1 output file written: test.1
Transcript written on test.log.
```

Labels or captions in a MetaPost drawing are often passed to TEX to process behind the scenes, as we shall see presently, and the result is a PostScript file we can include in our TEX in the ordinary way. It is assumed that the reader can find out how to do this.

Rather than showing the trivial result of that test, let us consider a slightly more sophisticated real graph (using data from the Protestant Cemetery, Rome—see Rahtz (1988)) which looks like this (henceforth we only show the MetaPost code between `begingraph` and `endgraph`):

```
draw begingraph(2.5in,1.75in);
gdraw "yearm.dat" dashed evenly;
gdraw "yearw.dat";
glabel.lft
 (btex (solid) Women etex, 1960,30);
glabel.lft
 (btex (dashed) Men etex ,1870,30);
glabel.bot
 (btex Number of burials per year
 ($n \approx 4300$) etex,OUT);
endgraph;
```

0-2-1

This shows some of the main features of the graph package for plotting data from external data files and labeling. The command `gdraw` (which can be used several times in succession) is followed by a file name; it reads data values (two per line, giving an *x* and *y* coordinate) from that file, and plots the resulting line. The effect can be varied with various modifiers — here we used `dashed evenly`. The command `glabel`, to place some captioning text, has a prefix (separated by .) which indicates where on the graph it is to go (`lft` = 'left', `bot` = 'bottom' etc). It is followed by an expression inside round brackets of text, an *x* coordinate, and a *y* coordinate. The special coordinate pair of `OUT` means it will be placed neatly outside the graph area. You can supply literal text in quotes, or have it processed by TEX by bracketing it with `btex` ... `etex` (no quotes around the text in this case).

The `graph` package can take care of:

- automatic scaling of data;
- automatic generation and labeling of tick marks or grid lines;
- multiple coordinate systems in the same picture;
- linear and logarithmic scales;
- plotting with arbitrary symbols;
- handling multiple columns in the same data file, with user-specified procedures.

3 Variations in basic graphing

If `gdraw` is followed by a `plot` command, a symbol can be drawn at each coordinate instead of a continuous line; the symbols is technically a MetaPost "picture", i.e. in practice some text which can be typeset by TEX, as the following variation shows:

```
draw begingraph(2.5in,1.75in);
gdraw "yearm.dat"
 plot btex $\bullet$ etex;
gdraw "yearw.dat"
 plot btex $\circ$ etex;
glabel.bot
 (btex Burials etex,OUT);
glabel.lft
 (btex Number etex rotated 90,OUT);
endgraph;
```

0-3-1

For this graph we also rotated the label for the *y* axis by 90° using a modifier to `btex` ... `etex`.

Frames, ticks, grids and scales

By default, graphs have a frame on all sides, no grid, and tick marks on the bottom and left. The frame can be altered with the `frame` command, which has a set of optional suffixes. Grid lines and ticks are controlled with `autogrid`:

`autogrid`(*x specification*,*y specification*)

The specifications can have the values `grid`, `itick` or `otick`, which produce grid lines, inner ticks, or outer ticks; they can be suffixed with `.top` or `.bot` for the *x* axis and `.lft` and `.rt` for the *y* axis, as the following example shows:

```
draw begingraph(2.5in,1.75in);
gfill "yearw.dat" withcolor red;
autogrid(grid.bot,itick.rt)
  withcolor .5white;
frame.llft;
endgraph;
```

0-3-2

To override graph's choice of where to put tick marks and how to write labels, you can add explicit ticks with `itick` or `otick` and grid lines with `grid`. These have the same suffixes as `autogrid` and are followed by a MetaPost picture variable containing a label or a `format` command, and a coordinate. `format` is used to control how numbers are printed:

`format`(*specification*,*number*)

The *specification* consists of an optional initial string, a percent sign, an optional number indicating precision (default 3), a conversion letter (`e`, `f` or `g`) and an optional final string. The conversion letter determines whether or not scientific notation is used; `%g` will use decimal format for most numbers. How the scientific notation used by `format` is typeset depends on a MetaPost macro called `init_numbers` (see manual); since this uses the `btex...etex` system, you may need to look at it carefully if you are concerned about precisely which fonts are used.

The next graph shows both types of explicit labeling; we have to remember to turn off the normal marks at the end!

```
draw begingraph(2.5in,1.75in);
gfill "yearw.dat" withcolor red;
for y=10,20,30:
  itick.lft(format("%g",y),y);
endfor
otick.top("19th century",1850);
otick.top("20th century",1950);
frame.llft;
autogrid(,);
endgraph;
```

0-3-3

The labeling can also be changed by `setcoords`

`setcoords`(*x style*,*y style*)

The parameters for *x* and *y* can be set to `log`, `-log`, `linear`, or `-linear`.

While the program's scaling of data to fit the graph usually gives the right results, it can be overridden with `setrange`:

`setrange`(*min*,*max*)

You need to supply the minimum and maximum coordinates. The special constant `origin` is a useful shorthand for (0,0). To leave any value to be figured out by MetaPost, specify `whatever`. If you specify no range at all, MetaPost works it out from the data values and adds a small border.

Reading data files

Although the `gdraw` and `gfill` commands often suffice, we can get more control over the data read from a file by using `gdata`:

`gdata`(*filename*, *variable*, *commands*)

The *commands* are executed for every line of data in *filename*, with the values for each column available as, *e.g. c*1, *c*2 . . . *c*n for the variable name *c*. *filename* is a META string, so simple names should be enclosed in quotes (file names can also be computed from META variables.) Using some more data from the Protestant Cemetery in which each line consists of a person's age at death, we can show the distribution of mortality by age by accumulating data in an array and using that to create a path:

```
draw begingraph(2.5in,1.5in);
numeric p[]; path r;
for j := 0 upto 100: p[j]:=0; endfor
gdata ("ages.dat",y, age:=(scantokens y1);
 p[age]:=p[age] + 1;);
r:=(0,0)
 for j := 1 upto 100: --(j,p[j]) endfor;
gdraw r;
frame.llft;
endgraph;
```


The only complications are the need to initialize the array and the conversion of the string representation read from the data file into a numeric value with `scantokens`.

When `gdata` reads data files, it stops when it reaches a blank line or end of file; if you start `gdata` again with the same file name, it carries on reading another set of data. This allows you to put all your data sets in one file, but use it with care. One problem is that data files remain open if there is a blank line at the end, since MetaPost thinks some more data might follow; if you have many small data files, this situation can cause a MetaPost error—check the end of your files.

This display in the example above is not very readable; it might be better to accumulate data per decade of death from the file. As this gets a little more complicated, we abstract the job into a MetaPost macro called by the `gdata` command:

```
draw begingraph(2.5in,1.75in);
setrange(origin,(100,100));
numeric p[]; path r;
for j := 0 step 10 until 100:
  p[j]:=0; endfor
def check(expr age) =
 if age < 100:
   xage:=round(age/10) * 10;
   p[xage]:=p[xage] + 1; fi
enddef;
gdata ("ages.dat",y,
  check((scantokens y1)););
r:=(0,0) for j := 0 step 10  until 100:
   --(j,p[j]) endfor --(100,0);
gfill r -- cycle withcolor blue;
frame.llft;
endgraph;
```


It is often useful to accumulate points on a path for each line read from the data file; the macro `augment` is provided for this. Given a suffix of a variable name of type "path" and a parameter of a coordinate, `augment` creates the path if it does not exist or adds the point to an existing path. We use this to show the gravestone data again, this time processed to provide separate figures of deaths per decade for women (column 2) and men (column 3):

```
  1800 3 6
  1810 9 15
  1820 26 64
  1830 31 88
  ...
```

For each decade, we keep track of the last point reached and augment separate paths for male and female; these are then shaded in different colors to show how the male and female patterns vary over time. We need to know the last

decade in order to establish a sensible corner for the filled shape. The female pattern appears as a dotted line on top of the male shading.

```
path m,w,last;
draw begingraph(3in,2in);
setrange((1800,0),(whatever,whatever));
gdata ("decade.dat",y,
  last:=((scantokens y1),0);
  augment.w(y1,y2);
  augment.m(y1,y3););
gfill (1800,0)--w--last--cycle
  withcolor red;
gfill (1800,0)--m--last--cycle
  withcolor green;
pickup pencircle scaled 3pt;
gdraw w dashed withdots;
pickup pencircle scaled .75pt;
glabel.bot (btex Number of burials per decade
     ($n \approx 4300$) etex,OUT);
endgraph
rotated 90;
```


0-3-6

The example demonstrates that the graph macros return a METAFONT picture that can then be transformed (in this case rotated).

Different graph types

With a little effort, graph can draw bar charts; to demonstrate this, we copy a chart from Goossens et al. (1994), p. 287, that was made with the LaTeX bar package. Our technique is to make a single path out of all the bars and fill the result at the end:

```
path s; numeric x,y;
draw begingraph(2.5in,1.75in);
gdata ("students.dat",c,
 x:=(scantokens c1) * 12;
 y:=(scantokens c2);
 augment.s((x-5,0)--
 (x-5,y)-- (x+5,y)--
 (x+5,0));
 if y < 0: glabel.top(c2,(x,0)); fi
 if y > 0: glabel.bot(c2,(x,0)); fi
);
gfill s--cycle withcolor .5white;
frame.llft;
endgraph;
```


0-3-7

We explicitly work out the corners of each bar and allow for their width by multiplying the *x* values by 12; the bars themselves span 5 units on either side of the data point, so there is a gap of 2 units between each one.

A similar technique is used in the next chart which shows the number of pages in chapters of *The LaTeX Graphics Companion*; this time we draw each bar separately, so that they can be shaded according to the values. The work is delegated to a macro, which also prints a rotated label for each bar. Because explicit *x* labels are supplied, labeling of the *x* axis is suppressed.

```
path m; numeric n,width;
width:=20; defaultscale:=0.6; n:=0;
def bar(expr name,value) =
 gfill(n,0)--(n,value)--
 (n+width,value)--(n+width,0)--cycle
 withcolor (value/100,value/100,value/100);
 picture p;
 p = name infont defaultfont
     scaled defaultscale rotated 90;
 glabel.rt
```

```
   (image(unfill bbox p; draw p),(n,10));
 n:=n+width;
enddef;
draw begingraph(2.5in,1.75in);
setrange((0,0),(11*width,100));
autogrid(,otick.lft);
gdata("chap.dat",c,bar(c1,(scantokens c2))););
endgraph;
```

0-3-8

The string value read from the first data column is put into a MetaPost picture variable by using the low-level command `infont`. This lets us use `bbox` technique to give the extent of the text, which is made white with `unfill`. `image` is a useful macro that yields the picture resulting from a sequence of drawing commands; we use that as a label. The data for this graph starts as follows:

```
  graphics 28
  stdgraph 26
  xypic 28
  mf 26
  ...
```

We can also present our earlier "decade" data as a dual bar chart, with male and female figures side by side. To do this we maintain two separate paths, fill one and leave the other as an outline:

```
path m[],w[];
def wcheck(expr decade,value) =
 augment.w1(decade,0);
 augment.w1(decade,value);
 augment.w1(decade+5,value);
 augment.w1(decade+5,0);
enddef;
def mcheck(expr decade,value) =
 augment.m1(decade+5,0);
 augment.m1(decade+5,value);
 augment.m1(decade+10,value);
 augment.m1(decade+10,0);
enddef;
draw begingraph(3.75in,2in);
gdata ("decade.dat",y,
  wcheck((scantokens y1),(scantokens y2));
  mcheck((scantokens y1),(scantokens y3)););
gfill m1--cycle;
gdraw w1;
glabel.bot (btex Number of burials per decade
     ($n \approx 4300$) etex,OUT);
frame.llft;
endgraph rotated 90;
```

0-3-9

With care, we can even draw pie charts using similar ideas. The following example reads data about gravestones in the Protestant Cemetery in the following form:

```
  Romanian 1 0.02796420582
  Czech 2 0.05592841163
  .....
  Italian 391 10.93400447
  German 508 14.20581655
  unknown 599 16.75055928
  English 1462 40.8836689
```

Here the second column is the number of gravestones per nationality and, to make the code less complicated, the third column is the percentage of the total. For each pie wedge, we use the `buildcycle` macro to find the smallest enclosed shape from the union of a whole circle and two lines extending from the center at the starting and closing angle of the segment. The fill color of the wedge is derived from the percentage.

```
numeric r,last; path c,w;
r:=5; c:=fullcircle scaled 2r;
last:=0.0;
def wedge (expr lang,value,perc) =
 numeric current,n,half,xoff,yoff;
 picture p;
 n:=perc*3.6;
 current:=last+n;    half:=last+(n/2);
 w:=buildcycle((0,0)--(2r,0) rotated last,
  c, (2r,0)--(0,0) rotated current);
 gfill w withcolor
   (0.8-(perc/100),0.8-(perc/100),0.8-(perc/100));
 gdraw w;
 if perc > 5:
 p = lang infont defaultfont
    scaled defaultscale;
 glabel(image(unfill bbox p; draw p),
  3/4r*dir(half));
 fi;
 last:=current;
enddef;
draw begingraph(3in,3in);
defaultscale:=0.7;
gdata ("langs.dat",c,
  wedge(c1, (scantokens c2),
   (scantokens c3)););
autogrid(,);  frame withcolor white;
endgraph;
```

0-3-10

The placement of the labels in the pie bears a little examination; they are placed in the center of each wedge, three quarters of way along the radius.

Another type of graph has a linear *x* scale and uses the *y* axis simply to compare sets of data. The following graph uses our cemetery data to show the first and last occurrences of each type of gravestone. The code is straightforward except that we draw the lines with a different sized pen (with square ends) and revert to a thin line to draw the scale and frame (only on the bottom, since the *y* axis is not linear).

```
draw begingraph(2.5in,2.5in);
n:=10;
defaultscale:=0.7;
pickup pensquare scaled 3pt;
setrange((1700,0),(whatever,whatever));
gdata("stones.dat", s,
gdraw ((scantokens s2),n)--
  ((scantokens s3),n);
glabel.lft(s1,(scantokens s2)-3,n);
n:=n+16;);
pickup pensquare scaled .5pt;
frame.bot;
autogrid(otick.bot,);
endgraph;
```

0-3-11

The data, ranked in order of first occurence, starts like this:

```
  Chest 1738 1966
  Head 1765 1986
  Column 1766 1960
  Plaque-on-base 1775 1986
  Pedestal 1786 1967
  Plaque-in-ground 1794 1985
```

Our last example is more unusual. We want to plot data from a survey grid and shade each grid square according to its data value; in the data file the first two columns are the coordinates of the lower left corner of the grid square, the third column is the absolute data value, and the fourth column is a percentage version:

```
  2 1 102 85
  2 2 10 98
  2 3 110 84
  2 4 112 83
  2 5 114 83
  ...
```

The text is printed in white or black depending on the percentage.

```
def sq(expr x,y,num,perc) =
 gfill(x,y)--(x+10,y)--
  (x+10,y+10)--(x,y+10)--cycle
 withcolor (perc/100,perc/100,perc/100);
 glabel(num,(x+5,y+5))
   if perc < 50: withcolor white  fi;
enddef;
defaultscale:=0.7;
draw begingraph(70mm,80mm);
setrange((20,10),(110,110));
autogrid(,);
gdata ("pot.dat",c,
 sq((scantokens c1)*10,
 (scantokens c2)*10,
 c3, (scantokens c4)););
endgraph;
```

0-3-12

References

[1] Bentley, J. and Kernighan, B. 1984. *GRAP — a language for typesetting graphs*. Computing Science Technical Report 114, AT&T Bell Laboratories, Murray Hill, NJ.

[2] Goossens, M., Mittelbach, F. and Samarin, A. 1994. *The LATEX companion*. Reading, MA: Addison-Wesley.

[3] Hobby, J. D. 1992. *A user's manual for MetaPost*. Computing Science Technical Report 162, AT&T Bell Laboratories.

[4] Hobby, J. D. 1993. *Drawing graphs with MetaPost*. Computing Science Technical Report 164, AT&T Bell Laboratories.

[5] Rahtz, S. 1987. The Protestant Cemetery, Rome: a study undertaken under the auspices of the Unione Internazionale degli Istituti di Archeologia, Storia e Storia dell'Arte in Roma. *Opuscula Romana*, **16**, 149–167.

V The UK TEX Users' Group

edited by Peter Abbott
uktug-enquiries@tex.ac.uk

The 1996–97 UKTUG committee

R. Fairbairns	Chair
P. Abbott	Treasurer and Membership Secretary
D. P. Carlisle	Committee Secretary
M. Clark	Meetings Secretary

K. Bazargan; S. P. Q. Rahtz; M. D. Wooding.

Book Discounts for UKTUG members

We have arrangements with Addison-Wesley for their well-known TEX-related publications, and with International Thomson Publishing to supply any of the very excellent O'Reilly & Associates Inc. series of books to members.

The agreed list of books, together with the discounted (at least 20%) price, is distributed occasionally with *Baskerville*, but is always available from the Treasurer, Peter Abbott.

Please add £1.50 for the first book and 50p for each book after the first on the same order, for despatch to a single address

We are only allowed to offer this service to **current** members of the UK TEX Users' Group and/or members of TUG. Please send your order and cheque (in UK £) to Peter Abbott (address in *Baskerville* masthead). Make cheques payable to 'UKTUG' please. All books will be routed through UKTUG. *In all cases* please notify Peter Abbott by email, phone, fax or letter when books are delivered. This means that provided the book(s) are in stock, it will normally take at least a week from receipt of order to delivery of the book(s).

Obtaining TEX

From the network – CTAN

The UK TEX Archive on `ftp.tex.ac.uk` is part of the CTAN (Comprehensive TEX Archive Network) collaborating network of archives on the Internet organised by the TEX Users Group.

The CTAN archives run an enhanced *ftp* server which supports dynamic compression, uncompression, and archive creation options. Fetch the top-level file `README.archive-features` for information. The server also supports site-defined commands to assist you. Please read `README.site-commands` for a brief overview.

Please report any problems with CTAN archives via email to `ctan@urz.Uni-Heidelberg.de`.

The main directories which make up CTAN are listed below; readers are referred to Graham Williams' *TEX and LATEX Catalogue* which is available from CTAN as `help/Catalogue/catalogue.html`

biblio bibliography-related files, such as BIBTEX.
digests back issues of TEX-related periodicals
dviware contains the various `dvi`-to-whatever filters and drivers
fonts fonts, both sources and pre-compiled
graphics utilities and macros related to graphics
help overviews of the archive and the TEX system
info files and tutorials which document various aspects of TEX
indexing utilities and related files for indexing
language material for typesetting non-English documents
macros macros packages for TEX and style files
support programs which can be used in support of TEX