

Introduction to METAPOST

John D. Hobby

*Bell Labs, Lucent Technologies
600 Mountain Ave.
Murray Hill, New Jersey 07974*

Abstract

METAPOST is a picture-drawing language very much like METAFONT except with PostScript output. The language provides access to all major features of Level 1 PostScript and it has facilities for integrating graphics with typeset text.

This paper gives a brief overview of the METAPOST language and how it can be used. A few of the more interesting features are described in detail.

This is a revised version of a paper that appeared in *EuroTeX '92 Proceedings of the 7th European T_EX Conference*.

1. Introduction

Although METAFONT was originally designed as a font-making tool, many people have recognized that it is also a powerful graphics language. The problem is that METAFONT's output is in the form of bitmap images instead of graphics primitives. A diagram can sometimes be created in METAFONT and typeset as a single huge character, but this is cumbersome and makes it difficult to deal with textual labels. A good example of work along these lines appears in [5] and [9].

Another approach is to modify the METAFONT interpreter so that it outputs PostScript. Previous work along these lines presented in [1] and [11] has concentrated on producing PostScript fonts rather than graphics. Unlike these earlier systems, the METAPOST system involves the creation of a new language similar to METAFONT, but specifically designed for producing PostScript graphics. Preliminary comments on METAPOST appeared in [3].

Since METAPOST is based on the public-domain METAFONT source code given in [7], METAPOST has been able to inherit all the features of METAFONT that make it a powerful graphics language:

- The ability to store and manipulate coordinate pairs, straight and curved paths, coordinate transformations, pen shapes, and complete pictures.
- A Flexible and powerful mechanism for constructing smooth curves and straight lines.

- The ability to draw straight and curved lines of any thickness and to fill a region given its boundary.
- Mechanisms for solving linear equations so that geometric information can be specified in a largely declarative manner.
- A powerful macro facility that allows the language to be extended syntactically and semantically.
- Operators for intersecting curves, finding tangent lines, finding points on a curve that match a given tangent direction, and extracting subpaths.

In addition to these features, METAPOST allows pictures to contain text, dashed lines, clipping paths, and areas filled with gray or other colors. There are also data types for colors and recipes for dashed lines. In addition, there are important facilities for generating and manipulating typeset text. Readers familiar with other graphics languages such as Kernighan's *Pic* [6] and Wichura's *PicTeX* [10] will see that METAPOST is considerably more powerful.

Section 2 gives a general idea of what the language is like and what can be done with it. More detailed discussions of interesting features follow in Section 3. This includes Section 3.1 on integrating text and graphics, Section 3.2 on dealing with dashed lines, and Section 3.3 on drawing arrows. Finally, Section 4 deals with macro packages and Section 5 presents some concluding remarks.

2. Overview of the Language

METAPOST is a batch-oriented graphics language that achieves great power and flexibility by giving up some the ease of use found in interactive graphics editors such as MacDraw. A METAPOST user prepares an input file such as the one shown in Figure 3. Invoking the METAPOST interpreter produces an encapsulated PostScript output file that can be included in a \TeX document or viewed with a PostScript interpreter such as GhostScript. The input file in the figure has a single `beginfig...endg` block. There could be more such blocks, in which case each would produce a separate output file

```
beginfig(1);
draw (20,20)--(0,0)--(0,30)--(30,0)--(0,0);
endfig;
end
```



Figure 3: A METAPOST input file and the resulting output

Since this paper is not intended to be a user's manual, no attempt will be made to show the exact syntax used to create subsequent examples. Instead, we

concentrate on general concepts with the aim of showing what METAPOST can do. The interested reader can refer to [2, 4] for details.

Another thing METAPOST can do is draw curved lines. If points P_0, P_1, \dots, P_4 are as in Figure 4a, asking the interpreter to connect them in order produces the curve in Figure 4b.

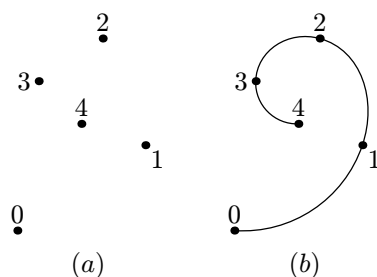


Figure 4: A sequence of points and a curve formed by connecting them.

Asking for a smooth closed curve through the same sequence of points produces Figure 5a. Since METAPOST has data types and operators for objects like curved lines, it is possible to store the curve in a *path variable* p and use a statement like

```
draw p scaled s
```

to draw rescaled versions of p . Figure 5b was generated by placing this statement in a loop that scans various values of s .

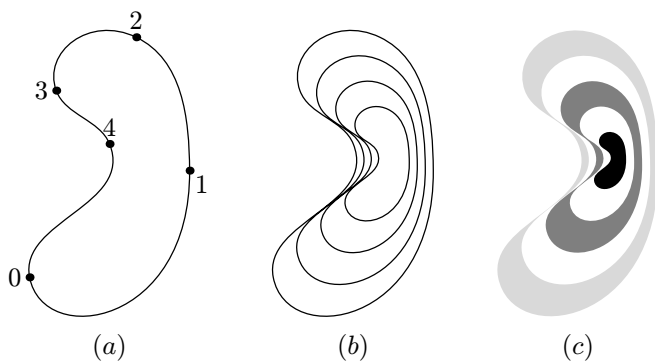


Figure 5: A closed curve and some effects that can be achieved by rescaling it.

There is also a *fill* statement that fills the interior of a closed curve with a color or a shade of gray. The filled regions in Figure 5c illustrate how overlapping fills overwrite each other. The figure was generated by filling the outermost

curve with light gray, then filling the next smaller curve with white, then the next smaller curve with dark gray, etc.

The examples given so far suggest that METAPOST allows drawing and filling, it has data types for numbers, coordinate pairs, and curved paths, it has operators for doing things like rescaling paths, and it has programming-language constructions such as loops. It also inherits from METAFONT the ability to solve linear equations and deal with a broad class of coordinate transformations.

Figure 6 illustrates linear equations and coordinate transformations. It was generated by introducing an unknown transformation τ , giving a pair of equations that declare it to be shape-preserving, and declaring that it maps Point 1 into Point 2 and Point 3 into Point 4. The figure was created by generating a simple picture P and repeatedly drawing P and transforming it by τ .

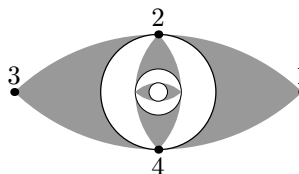


Figure 6: An example of repeated transformations.

3. Interesting Features

Anyone familiar with METAFONT can see that Section 2 did not begin to cover all the language features mentioned in the introduction. While it is impractical to give a detailed treatment of the entire language, we can concentrate on a few of the features that distinguish METAPOST from METAFONT and from other graphics languages.

3.1. Text in Pictures

METAPOST has a number of features for including labels and other text in the figures it generates. The simplest way to do this is to use the `label` statement to specify the label text and the point to be labeled. If you are labeling some feature of a diagram you probably want to offset the label slightly to avoid overlapping. This is illustrated in Figure 7 where statements of the form

```
label.top("a", <expression1>);
label.lft("b", <expression2>);
```

put the "a" label above the midpoint of the line it refers to and the "b" label to the left of the midpoint of its line. (In addition to `top` and `lft`, there are six other optional suffixes for other label positions.)

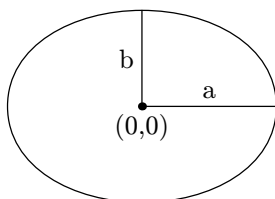


Figure 7: A labeled diagram.

There is also a `dotlabel` command that marks a point with a dot and positions text as the `label` command does. For instance, the command

```
dotlabel.bot("(0,0)", (0,0))
```

generates a dot marked "(0,0)" as in Figure 7.

For labeling statements such as `label` and `dotlabel` that use a string expression for the label text, the string gets typeset in a default font as determined by the string variable `defaultfont`. The initial value of `defaultfont` is likely to be "cmr10", but it can be changed to a different font name by giving an assignment such as

```
defaultfont:="ptmr8r"
```

When you change `defaultfont`, the new font name should be something that `TEX` would understand since `METAPOST` gets height and width information by reading the `tfm` file. (See [8]). It should be possible to use built-in PostScript fonts, but the names for them are system-dependent. The "ptmr8r" example is typically how one gets Times-Roman. (Note that `defaultfont` must not refer to a virtual font such as "ptmr".) A `TEX` font such as `cmr10` is a little dangerous because it does not have a space character or certain ASCII symbols. In addition, `METAPOST` does not use the ligatures and kerning information that comes with a `TEX` font.

The `METAPOST` language does not need elaborate typesetting abilities because there is a preprocessor that extracts `TEX` commands, runs them through `TEX` (or `LATEX`), and translates the output into a form that the interpreter understands. There is even a separate preprocessor that handles `troff` commands. Any time you say

```
btex <typesetting commands> etex
```

in a `METAPOST` input file, the preprocessor translates the `<typesetting commands>` into a `METAPOST` picture expression that can be used in a `label` or `dotlabel` statement. For instance, a statement of the form

```
label.lrt(btex $\sqrt{x}$ etex, <coordinates>)
```

was used to place the label \sqrt{x} in Figure 8.

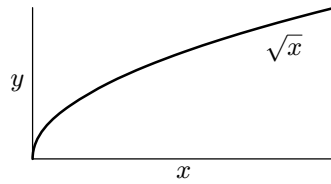


Figure 8: A figure with labels done in T_EX.

Figure 9 illustrates some of the more complicated things that can be done with labels. Since the result of `btex ...etex` is a picture, it can be operated on like a picture. In particular, it is possible to rotate the picture by giving

```
btex $y$ axis etex rotated 90
```

as the argument to a `label` statement.

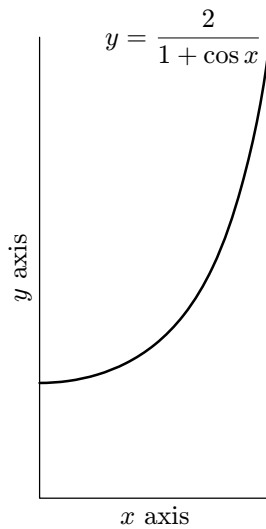


Figure 9: An example of how typeset labels can be rotated

Here is how T_EX material gets translated into a form METAPOST understands: The METAPOST processor skips over `btex ... etex` blocks and depends on a preprocessor to translate them into low level METAPOST commands. If the

main file is `g.mp`, the translated \TeX material is placed in a file named `g.mpx`. This is normally done silently without any user intervention but it could fail if one of the `btex ... etex` blocks contains an erroneous \TeX command. Then the erroneous \TeX input is saved in the file `mpxerr.tex` and the error messages appear in `mpxerr.log`.

\TeX macro definitions or any other auxiliary \TeX commands can be enclosed in a `verbatimtex ... etex` block. The difference between `btex` and `verbatimtex` is that the former generates a picture expression while the latter only adds material for \TeX to process. For instance, if you want \LaTeX to typeset labels using the `amstex` package, your METAPOST input file would look something like this:¹

```
verbatimtex \documentclass{article}
\usepackage{amstex} \begin{document} etex
beginfig(1);
  :
  label(btex <math> material using</math> etex, <coordinates>);
  :
```

METAPOST has an internal variable called `prologues` that controls the handling of text in pictures. Giving this internal variable a positive value causes output to be formatted as “structured PostScript” generated on the assumption that text comes from built-in PostScript fonts. This makes METAPOST output much more portable, but it generally does not work with \TeX fonts unless you have them in PostScript Type 1 format. Many `dvi-to-PostScript` programs download bitmaps for only those characters actually used in the document. Such programs can handle METAPOST output if they understand the nonstandard PostScript comments that the METAPOST interpreter uses to indicate which characters need to be downloaded. Modern versions of Rokicki’s `dvips` have this capability.

3.2. Dashed Lines

The METAPOST language provides many ways of changing the appearance of a line besides just changing its width. This is done by specifying a *dash pattern* when drawing a straight or curved line. Figure 10 shows a few examples of dash patterns and the lines they generate. There is a predefined dash pattern called `evenly` that makes dashes 3 points long separated by gaps of the same size. Another predefined dash pattern `withdots` produces dotted lines with dots 5 points

¹ It may be necessary tell the preprocessor to use \LaTeX instead of \TeX ; e.g., by setting environment variable `TEX=latex` on a Unix system.

instead of `draw <path expression>`. This draws the given path with an arrowhead at the end. If you want the arrowhead at the beginning of the path, there is an operator that reverses a path. For double-headed arrows, there is a `drawdblarrow` statement.

```

1 —————> 2 drawarrow z1..z2
3 ←————— 4 drawarrow reverse(z3..z4)
5 ←—————> 6 drawdblarrow z5..z6

```

Figure 12: Three ways of drawing arrows.

The size of the arrowhead is guaranteed to be larger than the line width, but it might need adjusting if the line width is very large. This is done by assigning a new value to the internal variable `ahlength` that determines arrowhead length as shown in Figure 13. Increasing `ahlength` from the default value of 4 PostScript points to 1.5 centimeters produces the large arrowhead in Figure 13. There is also an `ahangle` parameter that controls the angle at the tip of the arrowhead. The default value of this angle is 45 degrees as shown in the figure.

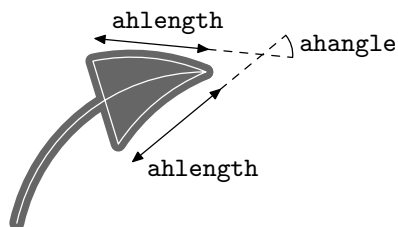


Figure 13: A large arrowhead with key parameters labeled and paths used to draw it marked with white lines.

The arrowhead is created by filling the triangular region that is outlined in white in Figure 13 and then drawing the boundary with the current line width. Readers familiar with METAFONT will recognize this as the `!draw` statement.

4. Macro Packages

This section describes auxiliary macros not included in Plain METAPOST. The macros described in Section 4.1 make it convenient to do things that *pic* is good at [6]. Section 4.2 makes some brief remarks about other macro packages. In order to use a macro package, it is necessary to give a METAPOST command that names the macro file and asks the interpreter to read it.

4.1. Macros for Boxes

The box-making macros are contained in a macro file called `boxes.mp`. This file can be accessed by giving the METAPOST command `input boxes` before any figures that use the box making macros.

The basic tool for making boxes is the command

$$\text{boxit}.\langle\text{box name}\rangle(\langle\text{picture expression}\rangle)$$

This creates variables $\langle\text{box name}\rangle.c$, $\langle\text{box name}\rangle.n$, $\langle\text{box name}\rangle.e$, ... that can then be used for positioning the picture before drawing it. The actual drawing is done by a separate command `drawboxed` that takes a list of box names.

If the command is `boxit.bb(\langle picture \rangle)`, the box name is `bb` and the contents of the box is the $\langle\text{picture}\rangle$. In this case, `bb.c` the position where the center of the picture is to be placed, and `bb.sw`, `bb.se`, `bb.ne`, and `bb.nw` are the corners of a rectangular path that will surround the picture. Variables `bb.dx` and `bb.dy` give the spacing between the picture and the surrounding rectangle, and `bb.o` is the amount by which the picture has to be shifted to achieve all this.

When the `boxit` macro is called with box name b , it gives linear equations that force $b.sw$, $b.se$, $b.ne$, and $b.nw$ to be the corners of a rectangle aligned on the x and y axes with the box contents centered inside as indicated by the gray rectangle in Figure 14. The values of $b.dx$, $b.dy$, and $b.c$ are left unspecified so that the user can give equations for positioning the boxes. If no such equations are given, macros such as `drawboxed` can detect this and give default values. The default values for `dx` and `dy` variables are controlled by the internal variables `defaultdx` and `defaultdy`.

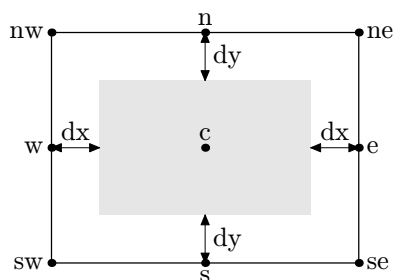


Figure 14: The relationship between the picture given to `boxit` and the associated variables. The picture is indicated by a gray rectangle.

If b represents a box name, `drawboxed(b)` draws the rectangular boundary of box b and then the contents of the box. This bounding rectangle can be accessed separately as `bpath b`, or in general

$$\text{bpath} \langle\text{box name}\rangle$$

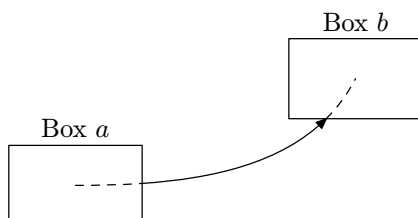


Figure 15: A “well-targeted arrow” generated by trimming the dashed sections from a curved path.

One interesting use of the bounding rectangle is for generating “well-targeted arrows” as shown in Figure 15. Given a path from the center of Box *a* to the center of Box *b*, there are METAPOST operators that make it convenient to chop off the parts of the path before the first intersection with `bpath a` and after the last intersection with `bpath b`.

There is also a special command

```
boxjoin(<equation text>)
```

that controls the relative position of consecutive boxes. Within the `<equation text>`, `a` and `b` represent the box names given in consecutive calls to `boxit` and the `<equation text>` gives equations to control the relative sizes and positions of the boxes. For example, the METAPOST code for Figure 16 uses

```
boxjoin(a.se=b.sw; a.ne=b.nw)
```

to causes boxes to line up horizontally. (It is instructive to compare this figure with the similar one in the `pic` manual [6]).

The `boxes.mp` macros also provide for circular and oval boxes. These are a lot like rectangular boxes except for the shape of the bounding path. Such boxes are set up by the `circleit` macro:

```
circleit<box name>(<picture expression>)
```

The `circleit` macro defines pair variable just as `boxit` does, except that there are no corner points `<box name>.ne`, `<box name>.sw`, etc. A call to

```
circleit.a(...)
```

gives relationships among points `a.c`, `a.s`, `a.e`, `a.n`, `a.w` and distances `a.dx` and `a.dy`. Together with `a.c` and `a.o`, these variables describe how the picture is centered in an oval as can be seen from the Figure 17.

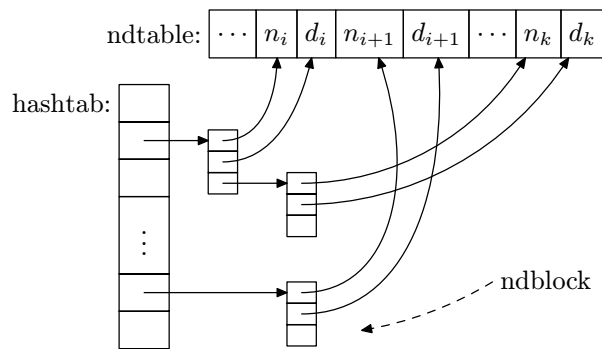


Figure 16: An example of what can be done with the `boxes.mp` macros

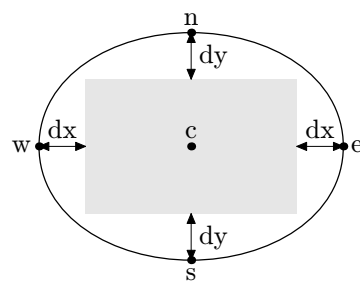


Figure 17: The relationship between the picture given to `circleit` and the associated variables. The picture is indicated by a gray rectangle.

The `drawboxed` and `bpath` macros work for `circleit` boxes just as they do for `boxit` boxes. By default, the boundary path for a `circleit` box is a circle large enough to surround the box contents with a small safety margin controlled by the internal variable `circmargin`. Figure 18 gives an example. The oval boundary paths around “Start” and “Stop” in the figure are due to equations of the form

$$\langle \text{box name} \rangle .dx = \langle \text{box name} \rangle .dy$$

that force those boxes to be noncircular.

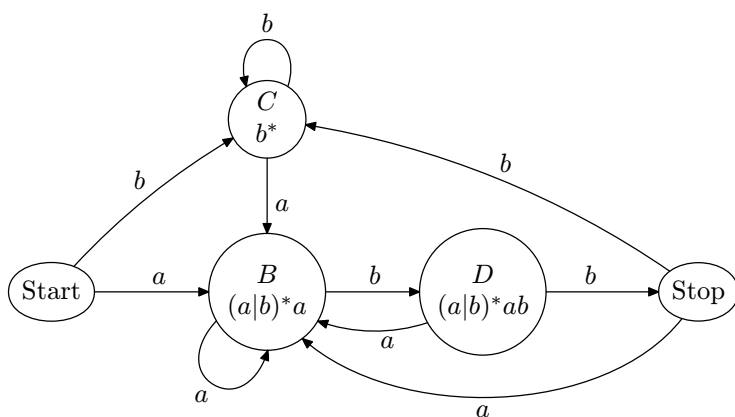


Figure 18: Circular and oval boxes generated using the `boxes.mp` macros.

4.2. Other Packages

Why aren’t the `boxes.mp` macros automatically preloaded like the plain macros? One reason is that they are too specialized to really be treated as part of the core language. Another reason is that `boxes.mp` was intended to be the first of several macro packages, each one extending the language to cover another specialized application.

In fact, there already is another macro package called `graph.mp`. It handles separate data files, automatic axis labeling, logarithmic scales, and other features [4]. There are also macro packages that add new box shapes to `boxes.mp` or provide additional features for use with `graph.mp`.

5. Conclusion

Building on METAFONT has made METAPOST a very powerful and flexible graphics language. It is especially well suited to generating figures in technical documents which may involve mathematical constraints that are best expressed symbolically. Such figures lack the aesthetic requirements that make font design so challenging.

This paper has introduced the METAPOST language via examples concentrating on interesting features that distinguish the language from other graphics languages and from METAFONT. Readers who want to use the language should refer to [2, 4]. The METAPOST interpreter is now widely available as part of many T_EX installations; e.g., the Web2C system has included METAPOST since version 7.0.

Bibliography

- [1] Leslie Carr. Of METAFONT and PostScript. In *T_EX User's Group Eighth Annual Meeting Conference Proceedings*. T_EX User's Group, Providence, Rhode Island, 1988.
- [2] J. D. Hobby. A user's manual for MetaPost. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, April 1992. Available as <http://cm.bell-labs.com/cs/cstr/162.ps.gz>.
- [3] John D. Hobby. A METAFONT-like system with PostScript output. *Tugboat, the T_EX User's Group Newsletter*, 10(4):505–512, December 1989.
- [4] John D. Hobby. Drawing graphs with metapost. Computing Science Technical Report no. 164, AT&T Bell Laboratories, Murray Hill, New Jersey, September 1993. Available as <http://cm.bell-labs.com/cs/cstr/164.ps.gz>.
- [5] Alan Jeffrey. Labelled diagrams in METAFONT. *TUGboat, Communications of the T_EX User's Group*, 12(2):227–229, June 1991.
- [6] Brian W. Kernighan. Pic—a graphics language for typesetting. In *Unix Research System Papers, Tenth Edition*, pages 53–77. AT&T Bell Laboratories, 1990.
- [7] D. E. Knuth. *METAFONT the Program*. Addison Wesley, Reading, Massachusetts, 1986. Volume D of *Computers and Typesetting*.
- [8] D. E. Knuth. *The T_EXbook*. Addison Wesley, Reading, Massachusetts, 1986. Volume A of *Computers and Typesetting*.

- [9] Richard O. Simpson. Nontraditional uses of METAFONT. In Malcom Clark, editor, *T_EX Applications, Uses, Methods*, pages 259–271. Ellis Horwood, 1990.
- [10] Michael J. Wichura. *The PiCT_EX Manual*. T_EX User's Group, Providence, Rhode Island, 1987.
- [11] Shimon Yanai and Daniel M. Berry. Environment for translating METAFONT to PostScript. *TUGboat, Communications of the T_EX User's Group*, 11(4):525–541, November 1990.