A beginner's guide to METAPOST for creating high-quality graphics

Troy Henderson
Department of Mathematical Sciences
United States Military Academy
West Point, NY 10996, USA
troy (at) tlhiv dot org
http://www.tlhiv.org

Abstract

Individuals that use TEX (or any of its derivatives) to typeset their documents generally take extra measures to ensure paramount visual quality. Such documents often contain mathematical expressions and graphics to accompany the text. Since TEX was designed "for the creation of beautiful books — and especially for books that contain a lot of mathematics" [4], it is clear that it is sufficient (and in fact *exceptional*) at dealing with mathematics and text. TEX was not designed for creating graphics; however, certain add-on packages can be used to create modest figures. TEX, however, is capable of including graphics created with other utilities in a variety of formats. Because of their scalability, Encapsulated PostScript (EPS) graphics are the most common types used. This paper introduces METAPOST and demonstrates the fundamentals needed to generate high-quality EPS graphics for inclusion into TEX-based documents.

1 Introduction

To accompany TEX, Knuth developed METAFONT as a method of "creating entire families of fonts from a set of dimensional parameters and outline descriptions" [1]. Approximately ten years later, John Hobby began work on METAPOST — "a powerful graphics language based on Knuth's METAFONT, but with PostScript output and facilities for including typeset text" [3]. Although several packages (e.g., PICTEX, XY-pic, and the native LATEX picture environment to name a few) are available for creating graphics within TEX-based documents, they all rely on TEX. Since TEX was designed to typeset text, it seems natural that an external utility should be used to generate graphics instead. Furthermore, in the event that the graphics require typeset text, then the utility should use TEX for this requirement. This premise is exactly the philosophy of METAPOST.

Since METAPOST is a programming language, it accommodates data structures and flow control, and compilation of the METAPOST source code yields EPS graphics. These features provide an elegant method for generating graphics. Figure 1 illustrates how METAPOST can be used programatically. The figure is generated by rotating one of the circles multiple times to obtain the desired *circular chain*. The programming language constructs of METAPOST also deliver a graceful mechanism for creating animations without having to manually create each frame of the animation. The primary advantage of EPS is that it can be scaled to any resolution without a loss in quality. It can also be easily converted to raster formats, e.g. Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG), et al., or other vector formats including Portable Document Format (PDF) and Scalable Vector Graphics (SVG), et al.

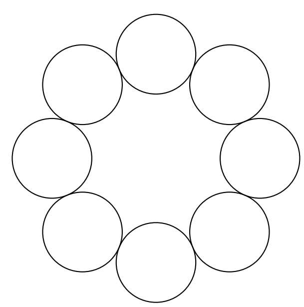

Figure 1: Rotated circles

[1] All graphics in this article (except Figure 2) are created with METAPOST, and the source code and any required external data files for each of these graphics are embedded as file attachments in the electronic PDF version of the article.

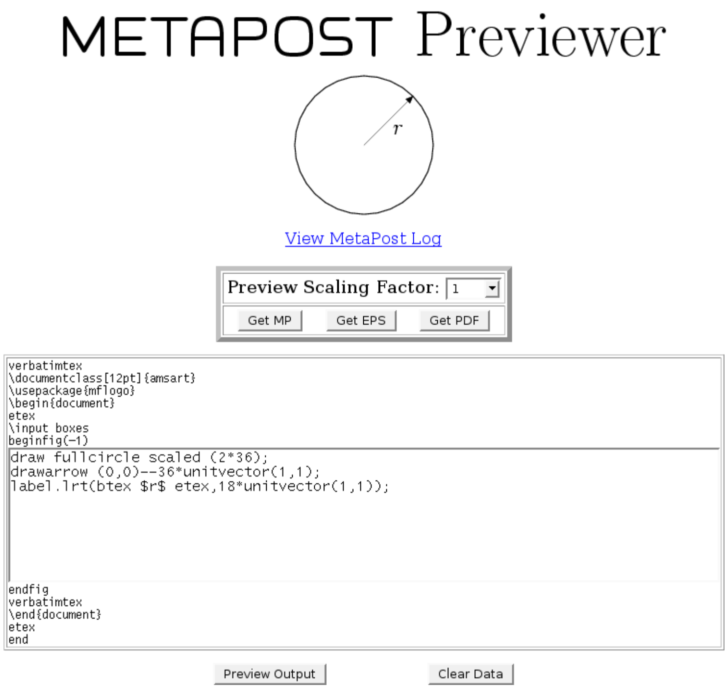

Figure 2: METAPOST Previewer

2 METAPOST compilation

A typical METAPOST source file consists of one or more figures. Compilation of the source file generates an EPS graphic for each figure. These EPS graphics are not self-contained in that fonts used in labels are not embedded into the graphic.

If `foo.mp` is a typical METAPOST source file, then its contents are of the following form:

```
beginfig(1);
   draw commands
endfig;
beginfig(2);
   draw commands
endfig;
...
beginfig(n);
   draw commands
endfig;
end;
```

Executing

```
mpost foo.mp
```

yields the following output:

```
This is MetaPost, Version ⟨version⟩
(foo.mp [1] [2] ... [n] )
n output files written: foo.1 .. foo.n
Transcript written on foo.log.
```

For users who just want to "get started" using METAPOST, a METAPOST previewer is available at `http://www.tlhiv.org/MetaPostPreviewer`. This previewer (illustrated in Figure 2) is simply a graphical interface to METAPOST itself. It generates a single graphic with the option to save the output in both EPS and PDF formats. Users may also choose to save the source code and can view the compilation log to assist in debugging.

3 Data types

There are nine data types in METAPOST: *numeric*, *pair*, *path*, *transform*, *color*, *string*, *boolean*, *picture*, and *pen*. These data types allow users to store fragments of the graphics for later use. We will briefly discuss each of these data types and elaborate on how they are used in a typical METAPOST program.

- *numeric* — numbers
- *pair* — ordered pairs of numerics
- *path* — Bézier curves (and lines)
- *picture* — pictures
- *transform* — transformations such as shifts, rotations, and slants
- *color* — triplets in the unit cube with red, green, and blue (RGB) components
- *string* — strings to be labeled
- *boolean* — "true" or "false" values
- *pen* — stroke properties

Virtually all programming languages provide a way of storing and retrieving numerical values. This is precisely the purpose of the *numeric* data type in METAPOST. Since graphics drawn with METAPOST are simply two dimensional pictures, it is clear that an ordered pair is needed to identify each point in the picture. The *pair* data type provides this functionality. Each point in the plane consists of an x (i.e., abscissa) part and a y (i.e., ordinate) part. METAPOST uses the standard syntax for defining points in the plane, e.g., `(`x`,`y`)` where both x and y are numeric data typed variables.

In order to store paths between points, the *path* data type is used. All paths in METAPOST are represented as cubic Bézier curves. Cubic Bézier curves are simply parametric splines of the form $(x(t), y(t))$ where both $x(t)$ and $y(t)$ are piecewise cubic polynomials of a common parameter t. Since Bézier curves are splines, they pairwise interpolate the points. Furthermore, cubic Bézier curves are diverse enough to provide a "smooth" path between all of the points for which it interpolates. METAPOST provides several methods for affecting the Bézier curve between a list of points. For example, piecewise linear paths (i.e., linear splines) can be drawn between a list of points since all linear polynomials are also cubic polynomials. Furthermore, if a specific direction for the path is desired at a given point, this constraint can be forced on the Bézier curve.

The *picture* data type is used to store an entire picture for later use. For example, in order to create animations, usually there are objects that remain the same throughout each frame of the animation. So that these objects do not have to be manually drawn for each frame, a convenient method for redrawing them is to store them into a picture variable for later use.

When constructing pairs, paths, or pictures in METAPOST, it is often convenient to apply affine transformations to these objects. As mentioned above, Figure 1 can be constructed by rotating the same circle several times before drawing it. METAPOST provides built-in affine transformations as "building blocks" from which other transformations can be constructed. These include shifts, rotations, horizontal and vertical scalings, and slantings.

There are five built-in colors in METAPOST: `black`, `white`, `red`, `green`, and `blue`. However, custom colors can be defined using the *color* data type. Colors in METAPOST are simply ordered triplets of the form `(`r`,`g`,`b`)` where r, g, and b are numerics between 0 and 1. These values r, g, and b identify what fraction of the color is red, green, and blue, respectively. For example, the built-in color `red` is simply a synonym for $(1, 0, 0)$ and `black` is a synonym for $(0, 0, 0)$. If a particular color is to be used several times throughout a figure, it is natural to store this color into a variable (of type *color*) for multiple uses.

The most common application of *string* data types is reusing a particular string that is typeset (or labeled). The *boolean* data type is the same as in other programming languages, used in conditional statements for testing. Finally, the *pen* data type is used to affect the actual stroke paths. The default unit of measurement in METAPOST is 1 bp = 1/72 in, and the default thickness of all stroked paths is 0.5 bp. An example for using the *pen* data type may include changing the thickness of several stroked paths. This new pen can be stored and then referenced for drawing each of the paths.

4 Common commands

The METAPOST manual [3] lists 26 built-in commands along with 23 function-like macros for which pictures can be drawn and manipulated using METAPOST. We will not discuss each of these commands here; however, we will focus on several of the most common commands and provide examples of their usage.

4.1 The `draw` command

The most common command in METAPOST is the `draw` command. This command is used to draw paths or pictures. In order to draw a path from `z1:=(0,0)` to `z2:=(54,18)` to `z3:=(72,72)`, we should first decide how we want the path to look. For example, if we want these points to simply be connected by line segments, then we use

```
draw z1--z2--z3;
```

However, if we want a smooth path between these points, we use

```
draw z1..z2..z3;
```

In order to specify the direction of the path at the points, we use the `dir` operator. In Figure 3 we see that the smooth path is horizontal at `z1`, a 45° angle at `z2`, and vertical at `z3`. These constraints on the Bézier curve are imposed by

```
draw z1{right}..z2{dir 45}..{up}z3;
```

Notice that `z2{dir 45}` forces the *outgoing* direction at `z2` to be 45°. This implies an *incoming* direction at `z2` of 45°. In order to require different incoming and outgoing directions, we would use

`draw z1{right}..{dir` θ_i`}z2{dir` θ_o`}..{up}z3;`

where θ_i and θ_o are the incoming and outgoing directions, respectively.

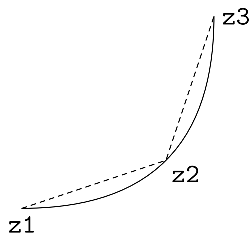

Figure 3: `draw` examples

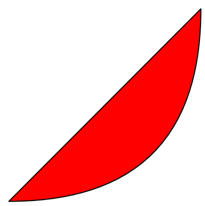

Figure 4: `fill` example

4.2 The `fill` Command

Another common command in METAPOST is the `fill` command. This is used to fill closed paths (or cycles). In order to construct a cycle, `cycle` may be appended to the path declaration. For example,

```
path p;
p:=z1{right}..z2{dir 45}..{up}z3--cycle;
fill p withcolor red;
draw p;
```

produces Figure 4. Notice that `p` is essentially the same curved path as in Figure 3 with the additional piece that connects `z3` back to `z1` with a line segment using `--cycle`.

Just as it is necessary to fill closed paths, it may also be necessary to *unfill* closed paths. For example, the annulus in Figure 5 can be constructed by

```
color bbblue;
bbblue:=(3/5,4/5,1);
path p,q;
p:=fullcircle scaled (2*54);
q:=fullcircle scaled (2*27);
fill p withcolor bbblue;
unfill q;
draw p;
draw q;
```

The `fullcircle` path is a built-in path that closely approximates a circle in METAPOST with diameter 1 bp traversed counter-clockwise. This path is not exactly a circle since it is parameterized by a Bézier curve and not by trigonometric functions; however, visually it is essentially indistinguishable from an exact circle. Notice that `p` is a `fullcircle` of radius 54 bp (3/4 in) and `q` is a `fullcircle` of radius 27 bp (3/8 in). The annulus is constructed by filling `p` with the baby blue color `bbblue` and then unfilling `q`. The `unfill` command above is equivalent to

```
fill q withcolor background;
```

where `background` is a built-in color which is `white` by default.

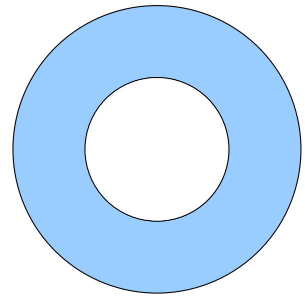

Figure 5: `unfill` example

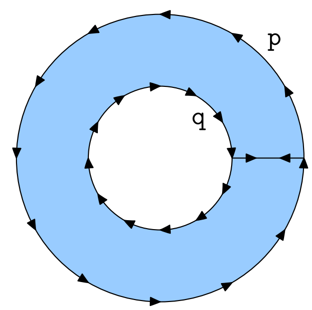

Figure 6: Avoiding an `unfill`

Often the `unfill` command appears to be the natural method for constructing figures like Figure 5. However, the `fill` and `unfill` commands in Figure 5 can be replaced by

```
fill p--reverse q--cycle withcolor bbblue;
```

The path `p--reverse q--cycle` travels around `p` in a counter-clockwise directions (since this is the direction that `p` traverses) followed by a line segment to connect to `q`. It then traverses clockwise around `q` (using the `reverse` operator) and finally returns to the starting point along a line segment using `--cycle`. This path is illustrated in Figure 6. One reason for using this method to construct the annulus as opposed to the `unfill` command is to ensure *proper transparency* when placing the figure in an external document with a non-white background. If the former method is used and the annulus is placed on a non-white background, say magenta, then the result is Figure 7. It may be desired to have the interior of `q` be magenta instead

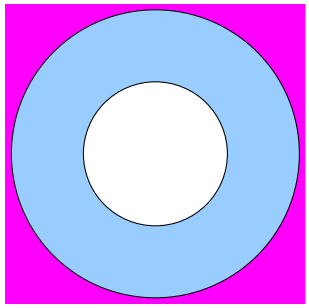

Figure 7: Improper transparency using `unfill`

of `white`. This could be accomplished by redefining `background`; however, the latter method described above is a much simpler solution.

4.3 Arrow commands

When drawing simple graphs and other illustrations, the use of arrows is often essential. There are two arrow commands in METAPOST for accommodating this need — `drawarrow` and `drawdblarrow`. Both of these commands require a path argument. For example,

```
drawarrow (0,0)--(72,72);
```

draws an arrow beginning at `(0,0)` and ending at `(72,72)` along the line segment connecting these points.

The path argument of both `drawarrow` and `drawdblarrow` need not be line segmented paths — they may be any METAPOST path. The only difference between `drawarrow` and `drawdblarrow` is that `drawarrow` places an arrow head at the end of the path and `drawdblarrow` places an arrow head at the beginning and the end of the path. As an example, to draw the curved path in Figure 3 with an arrow head at the end of the path (i.e., at `z3`), the following command can be used

```
drawarrow z1{right}..z2{dir 45}..{up}z3;
```

and is illustrated in Figure 8.

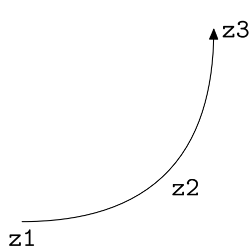

Figure 8: Using `drawarrow` along a path

4.4 The `label` command

One of the nicest features of METAPOST is that it relies on TEX (or LATEX) to typeset labels within figures. Almost all figures in technical documents are accompanied by labels which help clarify the situation for which the figure is assisting to illustrate. Such labels may include anything from simple typesetting as in Figures 3, 6, and 8 to typesetting function declarations and even axes labeling.

The `label` command requires two arguments — a string to typeset and the point for which label is placed. For example, the command

```
label("A",(0,0));
```

will place the letter "A" at the coordinate `(0,0)` and the box around this label is centered vertically and horizontally at this point. Simple strings like `"A"` require no real typesetting to ensure that they appear properly in the figure. However, many typeset strings in technical figures require the assistance of TEX to properly display them. For example, Figure 9 is an example where typesetting is preferred.

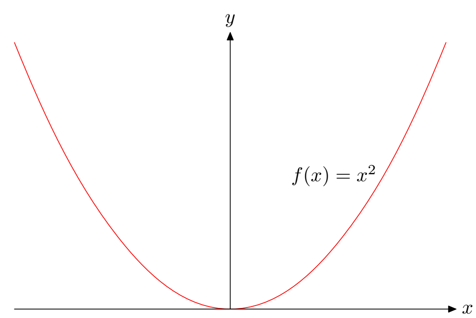

Figure 9: Labeling text

That is, the axes labels and the function declaration look less than perfect if TEX is not used. For reasons such as this, METAPOST provides a way to *escape* to TEX in order to assist in typesetting the labels. Therefore, instead of labeling the "A" as above,

```
label(btex A etex,(0,0));
```

provides a much nicer technique for typesetting the label. The `btex` ... `etex` block instructs METAPOST to process everything in between `btex` and `etex` using TEX. Therefore, the function declaration in Figure 9 is labeled using

`label(btex $f(x)=x^2$ etex,(`a,b`));`

where (a, b) is the point for which the label is to be centered.

Since many METAPOST users prefer to typeset their labels using LATEX instead of plain TEX, METAPOST provides a convenient method for accommodating this, done in the preamble of the METAPOST source file. The following code ensures that the `btex...etex` block escapes to LaTeX (instead of plain TeX) for text processing.

```
verbatimtex
%&latex
\documentclass{minimal}
\begin{document}
etex
beginfig(n);
  ⟨draw commands⟩
endfig;
end
```

Often times it is desirable to typeset labels with a justification that is not centered. For example, one may wish to place an "A" not centered horizontally about `(0,0)` but placed above `(0,0)`. METAPOST provides eight suffixes to accommodate such needs. The suffixes `.lft`, `.rt`, `.bot`, and `.top` align the label on the left, right, bottom, and top, respectively, of the designated point. A hybrid of these four justifications provide four additional ones, namely, `.llft`, `.ulft`, `.lrt`, and `.urt` to align the label on the lower left, upper left, lower right, and upper right, respectively, of the designated point. For example,

```
label.top(btex A etex,(0,0));
```

places the "A" directly above `(0,0)`. Figure 10 demonstrates each of the suffixes and their corresponding placement of the labels.

Figure 10: Label suffixes

5 Graphing functions

Among the most common types of figures for TeX users are those which are the graphs of functions of a single variable. Hobby recognized this and constructed a package to accomplish this task. It is invoked by

```
input graph;
```

METAPOST has the ability to construct data (i.e., ordered pairs) for graphing simple functions. However, for more complicated functions, the data should probably be constructed using external programs such as MATLAB (or Octave), Maple, Mathematica, Gnuplot, et. al.

A typical data file, say `data.d`, to be used with the `graph` package may have contents

```
0.0    0.0
0.2    0.447214
0.4    0.632456
0.6    0.774597
0.8    0.894427
1.0    1.0
```

This data represents the graph of $f(x) = \sqrt{x}$ for six equally spaced points in $[0, 1]$. To graph this data, the size of the graph must first be decided. Choosing a width of 144 bp and a height of 89 bp, a minimally controlled plot (as in Figure 11) of this data can be generated by

```
draw begingraph(144bp,89bp);
  gdraw "data.d";
endgraph;
```

The `graph` package provides many commands used to customize generated graphs, and these commands are fully documented in the manual [2] for the `graph` package.

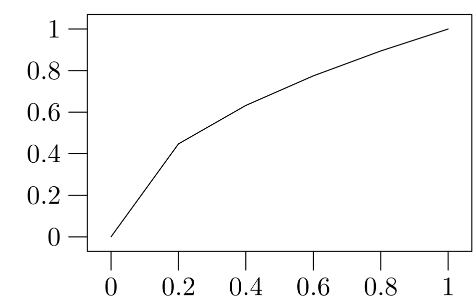

Figure 11: $f(x) = \sqrt{x}$ using the `graph` package

6 Including METAPOST figures in LaTeX

In order to include a METAPOST figure in LaTeX, the `graphicx` package is suggested. Below is an example of including a METAPOST figure (with name `foo.1`) in a LaTeX document.

```
\documentclass{article}
\usepackage{graphicx}
\usepackage{ifpdf}
\ifpdf
  \DeclareGraphicsRule{*}{mps}{*}{}
\fi
\begin{document}
...
\includegraphics{foo.1}
...
\end{document}
```

The ifpdf package and \ifpdf...\fi command is used to prompt PDFLaTeX to convert the METAPOST graphic to PDF "on the fly" using Hans Hagen's mptopdf. This conversion is necessary since PDFLaTeX performs no PostScript processing.

7 Conclusion

METAPOST is an elegant programming language, and it produces beautiful graphics. The graphics are vectorial and thus can be scaled to any resolution without degradation. There are many advanced topics that are not discussed in this article (e.g., loops, flow control, subpaths, intersections, etc.), and the METAPOST manual [3] is an excellent resource for these advanced topics. However, the METAPOST manual may seem daunting for beginners. There are many websites containing METAPOST examples, and several of these are referenced at http://www.tug.org/metapost. Finally, we mention that Knuth uses nothing but METAPOST for his diagrams.

References

[1] N. H. F. Beebe. Metafont. http://www.math.utah.edu/~beebe/fonts/metafont.html, 2006.

[2] J. D. Hobby. Drawing graphs with MetaPost. Technical Report 164, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992. Also available at http://www.tug.org/docs/metapost/mpgraph.pdf.

[3] J. D. Hobby. A user's manual for MetaPost. Technical Report 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992. Also available at http://www.tug.org/docs/metapost/mpman.pdf.

[4] D. E. Knuth. *The TeXbook*, volume A of *Computers and Typesetting*. Addison Wesley, Boston, 1986.