Using MFPIC with METAPOST

The MFPIC system now includes support for METAPOST. This necessitated rewriting so many of the GRAFBASE macros that a separate file, `grafbase.mp`, implements the METAPOST code while `grafbase.mf` still contains the METAFONT code. This document addresses the issues that arose with the added METAPOST support.

There were a great many changes to `mfpic.tex`, but I believe we have retained backward compatibility (with versions after 0.2.10.9) in the following senses:

- Previous files that input `mfpic.tex` can be processed with the new package. While they produce slightly different METAFONT code, the pictures should be the same except that some broken code has been fixed.
- Previous `.mf` files produced by TEXing the above files should still be correct and produce the same pictures as before when processed with METAFONT.
- Previously produced `.mf` files and new `.mf` files produced without turning on METAPOST support, can nevertheless be processed with METAPOST without error. This will not permit any of the PostScript specific enhancements, and may not produce exactly the same picture due to essential differences between METAFONT's and METAPOST's output format.

However, no backward compatibility is guaranteed for `\mfsrc` commands that write raw METAFONT or GRAFBASE code. Moreover there has been a change in the code, beginning around version 0.3.8, so that changes to most (all?) MFPIC parameters are local if they are changed inside an `mfpic` environment. Thus, files that rely on a change like `\axisheadlen{6pt}`, made in one picture, persisting to the next picture, will be broken. We think the new behavior is better, and more in line with user expectations.

There is no change to the usage and the new MFPIC macros write (nearly) the same METAFONT file as before. However the command `\usemetapost` will cause a `.mp` file to be written that will be significantly different than the `.mf` otherwise produced. Running METAPOST on this file will create EPS graphics files instead of a font file. The new MFPIC now includes support for including these graphics. Naturally, it is required that you have a system that allows printing of EPS figures. A typical system might consist of `epsf.tex` and/or the LaTeX2$_\varepsilon$ GRAPHICS package for graphics inclusion, plus the DVIPS driver, plus either a PostScript printer or Ghostscript. In addition, pdfTeX and pdfLaTeX can be used. See below for more details about the requirements.

1. Using METAPOST.

The simplest way to turn on METAPOST support is to issue the command

```
\usemetapost
```

in your source file. This must come after inputting the MFPIC macros, but before the `\opengraphsfile` command. In LaTeX2$_\varepsilon$ one can use `\usepackage[metapost]{mfpic}`. Changing between metafont and metapost options in a single document has not been tested and is officially unsupported.

To use MFPIC with METAPOST, the following support is needed (besides a working METAPOST installation):

Under plainTEX	The file `epsf.tex`
Under LATEX209	The file `epsf.tex` or `epsf.sty`
Under LATEX2$_\varepsilon$	The package GRAPHICS or GRAPHICX
Under pdfLATEX	The package GRAPHICS or GRAPHICX with option `pdftex`
Under plain pdfTEX	The files `supp-pdf.tex` and `supp-mis.tex`
In all cases	The files `grafbase.mp` and `dvipsnam.mp` plus, of course, `mfpic.tex` (and `mfpic.sty` for LATEX)

The file `grafbase.mp` should be in a directory searched by METAPOST. The remaining files should be in directories searched by the appropriate TEX variant. At present only versions of these files and packages current at the time of testing have been tested. If METAPOST cannot find the file `grafbase.mp`, then by default it will try to input `grafbase.mf`, with generally fatal results.

In case pdfLATEX is used, the graphics package should be given the `pdftex` option. This option requires the files `pdftex.def`, `supp-pdf.tex` and `supp-mis.tex`. The first of these is supplied with the GRAPHICS package, and the other two are usually supplied with a pdfTEX distribution.

If the user explicitly inputs one of the above required files or packages before the MFPIC macros are loaded then MFPIC will not reload them. If they have not been input, MFPIC will load whichever one it decides is required. In the LATEX2$_\varepsilon$ case, MFPIC will load the GRAPHICS package. If the user wishes GRAPHICX, then that package must be loaded before MFPIC. As a convenience, any options to MFPIC other than the ones it recognizes are passed on to the GRAPHICS package before loading it. Therefore, the following code

```
\usepackage[dvips,metapost]{mfpic}
```

will load the MFPIC macros, then load the GRAPHICS package with the `dvips` option. On the other hand,

```
\usepackage[dvips]{graphicx} \usepackage[metapost]{mfpic}
```

is required to use the GRAPHICX package with the DVIPS driver option.

If METAPOST support is turned on, the user is expected to run METAPOST on the resulting `.mp` file instead of METAFONT. A typical sequence of operations is as follows.

1. The user creates some file `user.tex` which contains one of the above methods of invoking METAPOST support, the command `\opengraphsfile{thefigs}`, and then some `\mfpic` commands or `mfpic` environments (followed later by `\closegraphsfile`, of course).
2. The user runs the appropriate variant of TEX on `user.tex`. If all goes without error, the file `thefigs.mp` is created.
3. The user runs METAPOST on the file `thefigs.mp`. METAPOST inputs `grafbase.mp` containing METAPOST macros to aid in creating files `thefigs.1` through `thefigs.N` (where `N` is the number of figures) each containing a different EPS figure.
4. The user runs TEX again. It is at this stage that the figure inclusion is done by the appropriate macros from `epsf.tex` or the GRAPHICS package, or the `supp-pdf.tex`.

5. The user runs the appropriate DVI-to-PS driver such as DVIPS, on `user.dvi`, and then prints or views the resulting `user.ps` using, e.g., GHOSTSCRIPT. If pdfTEX or pdfLATEX is used in step 4, a `.pdf` file is output instead of `.dvi`. In this case the user prints or views it with a PDF viewer.

2. Additions.

Invoking METAPOST support adds certain extensions to MFPIC.

2.1 *Color support*

The code in `grafbase.mp` keeps track of six colors:

`drawcolor`, `fillcolor`, `hatchcolor`, `headcolor`, `tlabelcolor` and `background`.

These control the colors used for drawing lines and curves, for filling regions, for hatching the interiors of regions, for drawing arrowheads, and for text that is placed by METAPOST (requires the mplabels option). The easiest way to invoke some color other than black (the default) is via the following MFPIC macros:

`\drawcolor{`⟨*color*⟩`}`
`\fillcolor{`⟨*color*⟩`}`
`\hatchcolor{`⟨*color*⟩`}`
`\headcolor{`⟨*color*⟩`}`
`\tlabelcolor{`⟨*color*⟩`}`
`\backgroundcolor{`⟨*color*⟩`}`

These commands write code into the `.mp` file that sets the value of the color for the appropriate figure elements. The macro `\gfill` uses the `fillcolor`. When `\ifpointfill` tests true then also `\point` draws points in `fillcolor`. The new macro `\polkadot`, draws the polkadots in `fillcolor`. All the hatching commands `\xhatch`, `\rhatch`, etc., use `hatchcolor`. All drawing of paths, curves, lines, etc., including dashed and dotted lines, the shafts of arrows and the boundary of unfilled points use `drawcolor`. All arrowheads are drawn in `headcolor` including those created by the `\axes` command. All text placed by `\tlabel` (and similar macros) when mplabels is in force will be in `tlabelcolor`. `\gclear` uses the color `background` (POSTSCRIPT doesn't permit removal of ink, so filling with the background color simulates clearing a region). Also the interior of unfilled points is colored `background`. All colors are initialized to black except `background` is white.

The `\shade` command is rather obsolete, but for backward compatibility, it is accepted, but translated to a filling with gray. The shade of gray is calculated from the values of `\shadespace` and `\shadewd`, and the default values produce `0.75*white`.

A ⟨*color*⟩ is one of the predefined color names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta`, or `yellow` or any legal METAPOST color expression. These expressions resolve to a triplet of numbers separated by commas and enclosed in parentheses. The numbers will be truncated to lie between 0 and 1. The predefine colors `black`, `white`, `red`, `green`, and `blue` resolve respectively to `(0,0,0) (1,1,1) (1,0,0) (0,1,0) (0,0,1)`. It is legal to add colors and multiply them by numbers. Therefore, `\fillcolor{blue+green}` is legal and is essentially the definition of `cyan`. The other predefined colors are `magenta = red + blue` and `yellow = red + green`. Also valid is `\fillcolor{0.7*white}`, which gives

a medium gray. In a pinch, numbers may be used directly, as in `\fillcolor{(1,.5,0)}`, which should give some sort of orange (depending on your display or printing device). The color `(1,0.5,0)` is the same as `0.5*red + 0.5*yellow`. You can darken any color by multiplying it by a number less than 1.

As a convenience several grafbase macros allow specifying color using cmyk, RGB, graylevels, or predefined color names. Thus, all of the following are valid MFPIC color specifications: `cmyk(1,0.50,0,0)`, `rgb(0,.5,1)`, `RGB(0,127.5,255)`, `RoyalBlue` (the last is defined in `dvipsnam.mp`; all produce the same color). The color specification `gray(.7)` is the same as `0.7*white`. Use these, for example, as follows

```
\headcolor{RoyalBlue}
\fillcolor{cmyk(1,0.5,0.0)}
```

If these color setting commands are used outside any `\mfpic` environment, they are global, and affect all following environments. If they are issued inside an environment, they affect only subsequent figure elements inside that environment.

If METAPOST support has not been selected, then these commands issue an error message, and MFPIC then proceeds as if nothing has happened.

For LaTeX2$_\varepsilon$ users, an alternative syntax is permitted similar to LaTeX2$_\varepsilon$'s color commands. For example:

`\fillcolor[`⟨*model*⟩`]{`⟨*color-spec*⟩`}`

where ⟨*model*⟩ is one of `rgb, RGB, cmyk, named,` or `gray`, and ⟨*color-spec*⟩ is a specification of the values appropriate to the model. For example:

```
\hatchcolor[cmyk]{0,.8,.6,.2}
```

This example is equivalent to `\hatchcolor{cmyk(0,.8,.6,.2)}`

If the `named` model is used, the name specified must be a previously defined color name, either one of the predefined names, or a name defined in `dvipsnam.mp`, or one defined by the following command:

`\mfpdefinecolor{`⟨*name*⟩`}{`⟨*model*⟩`}{`⟨*color-spec*⟩`}`

This will define a named color. Its use is best illustrated by an example, After

```
\mfpdefinecolor{DarkPuce}{rgb}{.8,.12,.56}
```

then one may use any of the following, with the same effect:

```
\drawcolor[named]{DarkPuce}
\drawcolor[rgb]{.8,.12,.56}
\drawcolor{rgb(.8,.12,.56)}
\drawcolor{DarkPuce}
```

2.2 *Color options*

For more control over color of individual elements, the macros `\gfill` and `\draw` have been given optional arguments for specifying the color. In addition `\point` and the hatching macros now have a second optional argument for specifying the color, and the `\arrow` macro has another optional argument (now four altogether) for determining the color of the head.

This optional color will be used for the single element only and has no effect on `fillcolor`, `drawcolor`, `hatchcolor`, or `headcolor`. See `mfpicdoc.tex` for the syntax of the color options.

Here is an example mixing these different methods of changing color:

```
\hatchcolor{green}
\draw[blue]\hatch\gfill[red]
\circle{(0,0),2}
```

This will draw a red filled, green hatched, blue circle. At the moment, the `\dashed` and `\dotted` commands do not have new color options. Use something like the following:

```
\drawcolor{blue}
\dotted\gfill[red]\circle{(0,0),2}
\drawcolor{black}
```

to fill a circle with red, and outline it with a dotted blue boundary, returning to a default black.

2.3 *Cautions.*

PostScript is not a pixel oriented language and so neither is METAPOST. The model for drawing objects is completely different between METAFONT and METAPOST, and so one cannot always expect the same results. METAPOST support in MFPIC was carefully written so that files successfully printed with MFPIC using METAFONT would be just as successfully printed using METAPOST. Nevertheless, it will almost certainly choke on files that make use of the `\mfsrc` command for writing code directly to the `.mf` file. While `grafbase.mp` is closely based on `grafbase.mf`, much of the code had to be completely rewritten.

Pictures in METAPOST are stored as (possibly nested) sequences of objects, where objects are things like points, paths, contours, other pictures, etc. In METAFONT, pictures are stored as a grid of pixels. Thus, pictures that are relatively simple in one program might be very complex in the other and even exceed memory allocated for their storage. A case in point is the `shade` command in `grafbase.mf`. The METAFONT code simply added regularly spaced dots (which each contained typically about 4 pixels at the resolution 300dpi) to the interior of a contour. Essentially the same code in METAPOST creates a picture containing an extremely large number of tiny circles, stored as filled cyclic contours with eight nodes each. Even modest sized regions with modest values of `\shadespace` overflowed the METAPOST capacity in two of the distributions tested. In addition, the resulting shading tended to be of very low quality when viewed or printed with the software we tested. Therefore, the `\shade` macro under the METAPOST option merely fills with a gray color calculated to be black if the shading space is not larger than the size of the dots. Keeping `\shade` at all was done purely for backward compatibility since `\gfill` now has a color option for METAPOST.

A model that helped me in developing the METAPOST macros for PostScript support is as follows. Think of each figure element as being painted on a transparent sheet. The final picture is obtained by laying each sheet on top of the previous sheets in the order they are drawn. In this model, the sheets are transparent, but the paint is opaque. This

leads to some problems. For example, paint cannot be removed*, it can only be covered up. Therefore the code for *unfilling* a region is really just painting over in the color `background`. This background color is still opaque, so if you create a picture of an annulus by filling a circle and then unfilling a smaller circle, the center "hole" is not really a hole. Placing this picture on top of another will obscure everything, nothing will show through the hole. In METAFONT, the aforementioned annulus can be obtained by simply zeroing the pixels in the smaller circle. Adding this on top of another picture will allow anything in the hole to show through.

Therefore, the order in which picture elements are added is of great importance. In `mfpic` environments, figures are drawn in the order they are written. But when prefix macros *add* elements to a curve, these elements are drawn in reverse order.† For example, `\rect{(0,0),(1,1)}` simply draws a square with line thickness equal to the default pen width. And `\gfill\rect{(0,0),(1,1)}` draws only the filled interior of that square. But `\draw\gfill\rect{(0,0),(1,1)}` fills the interior, and *then* draws the boundary. Part of the filled interior is covered by the pen stroke (the filling is done right up to the boundary, the pen is stroked with its center along the boundary). This usually gives the best picture when the interior and boundary are different colors. On the other hand, `\gfill\draw\rect{(0,0),(1,1)}` draws the boundary and *then* fills the interior. Part of the boundary stroke (half its thickness) is covered by the filling painted over it. This covering makes no difference when all the colors are the same (black), but now that colors are introduced it can make a big difference in the final picture. Placing prefixes in the following order (any prefix could be absent) should have most pictures looking best:

- Any arrow drawing prefixes first. Arrows are drawn with a simple `\arrow` or with `\arrow\reverse` or `\arrow\reverse\arrow`
- Any prefix for dashing, dotting or drawing the curve next. These include `\dashed`, `\dotted`, and `\draw`.
- Any hatching prefix or `\polkadot` next. If both are used, the one you want on top should be first.
- Any filling prefix next (`\shade` or `\gfill`).
- Any transformation prefix (such as `\rotatepath`) or closing prefix (like `\lclose`) next, immediately before the curve. The order of these is not important.
- The curve (or a set of curves in a `\connect` group) last.

The placement of transformation macros (like `\rotatepath`) depends on what effect you want to achieve. Something like

```
\draw\rotatepath{(0,0),45}\shade\rect{(0,0),(1,1)}
```

will shade the original rectangle, but draw the outline of the rotated version. If you want just one rectangle with various additions, put the transformation macro right before the figure macro.

---

* The `\gclip` command would seem to contradict this, but what METAPOST actually does with that command is to give the picture the attribute of being clipped. In the POSTSCRIPT output, a clipping path is defined, and the paint outside that path is never laid down. Still, one can think of this as the exception from the point of view of MFPIC.

† In programming terms, prefixes are *right associative*.

The definition of `\shade` as a gray fill means that old `.tex` files using MFPIC might need to be edited. The position of `\shade` among the prefixes might need to be changed, or a shaded figure might need to be drawn first so that other curves are not completely covered up.

2.4 *Text Manipulations*

If text is placed by METAPOST using the mplabels option, then text may be rotated. One might conceivably perform other transformtions, but that has not (yet) been implemented.

3. Other Considerations

It may be impossible to completely cater to all possible methods of graphic inclusions with automatic tests. The macro that actually causes the PostScript graphic to be included is `\setmfpicgraphic`, and the user may (carefully!) redefine this to suit special circumstances. The following are the default definitions.

In plain TeX: `\def\setmfpicgraphic#1{\epsfbox{#1}}`
In LaTeX209: `\def\setmfpicgraphic#1{\epsfbox{#1}}`
In LaTeX2$_\varepsilon$: `\def\setmfpicgraphic#1{\includegraphics{#1}}`
In pdfLaTeX: `\def\setmfpicgraphic#1{\includegraphics{#1}}`
In pdfTeX: `\def\setmfpicgraphic#1{\convertMPtoPDF{#1}{1}{1}}`

Moreover, since METAPOST by default writes files with numeric extensions, we add code to each figure, so that these graphics are correctly recognized as EPS or MPS. For example, to the figure with extension `.1`, we add the equivalent of one of the following

`\DeclareGraphicsRule{.1}{eps}{.1}{}` in LaTeX2$_\varepsilon$.

`\DeclareGraphicsRule{.1}{mps}{.1}{}` in pdfLaTeX.

Also, after running the command `\setmfpicgraphic`, MFPIC's figure placement code runs `\getmfpicoffset` to store the lower left corner of the bounding box of the figure in two macros `\mfpicllx` and `\mfpiclly`. All the above definitions of `\setmfpicgraphic`, except `\includegraphics`, make this information available, and the definition of merely copies it into these two macros. What MFPIC does in the case of `\includegraphics` is to modify (locally) the definition of an internal command of the graphics package so that it copies the information to those macros, and then `\getmfpicgraphic` does nothing. Changes to `\setmfpicgraphic` might require changing `\getmfpicoffset`

One possible reason for wanting to redefine `\setmfpicgraphic` might be to rescale all pictures. This is definitely not a good idea without the option mplabels since the MFPIC code for placing labels and captions and reserving space for the picture relies on the picture having the dimensions given by the arguments to the `\mfpic` command. With mplabels plus truebbox it will probably work, but (i) it has *not* been considered in writing the MFPIC code, (ii) it will then scale all the text as well as the figure, and (iii) it will scale all line thickness, which should be a design choice independent of the size of a picture. To rescale all pictures, one need only change `\mfpicunit` and rerun TeX and METAPOST.

A better reason might be to allow the conversion of your METAPOST figures to some other format. Then redefining `\setmfpicgraphic` could enable including the appropriate file in the appropriate format.

The argument of the `\setmfpicgraphic` command is the filename resulting from running the macro `\setfilename`. The command `\setfilename` gets two arguments: the name of the METAPOST output file (set in the `\opengraphsfile` command) without extension, and the number of the picture. The default definition of `\setfilename` merely inserts a dot between the two arguments. That is `\setfilename{fig}{1}` produces `fig.1`. You can redefine this behavior also. Any changes to `\setfilename` must come after the MFPIC macros are input and before the `\opengraphsfile` command. Any changes to `\setmfpicgraphic` must come after the MFPIC macros are input and before any `\mfpic` commands, but it is best to place it before the `\opengraphsfile` command.

As a hypothetical example, let us say you run TEX on your source file, producing `fig.mp`, and then you run METAPOST, producing `fig.1` through `fig.10`. Now you convert them to some hypothetical format `fig-1.pxy` through `fig-10.pxy`. You could then do something like the following:

```
\def\setfilename#1#2{#1-#2.pxy}
\def\setmfpicgraphic#1{\includepxy{#1}} \def\getmfpicoffset#1{< code >}
```

and the figures will be automatically included (assuming that the hypothetical command `\includepxy` can include the hypothetical graphics format `pxy` and that $<$ *code* $>$ saves the lower left corner of the bounding box of `.mp` in the appropriate macros). However you define `\setfilename`, it must produce the result without assignments or side effects. (In technical terms it must be *completely expandable.*) Something that is very difficult without assignments is to change the number, like trying to get `\setfilename{file}{`$\langle n \rangle$`}` to produce `file.`$\langle n+1 \rangle$. I don't recommend trying this. Instead, use `\mfcpicnumber{2}` before the `\opengraphsfile` command to start the figure numbering from 2.

Some figure inclusion macros (such as `\epsfbox` from `epsf.tex`) generate rather large amounts of error messages for nonexistent figure files, so MFPIC tests for the existence of the file before attempting an inclusion. Therefore, `\setfilename{#1}{#2}` should produce the actual name of the necessary file when `#1` is the name supplied in the `\opengraphsfile` command and `#2` is the number of the `mfpic` environment.

The code in MFPIC detects the TEX format in the following way: If `\documentclass` is a defined command, LATEX2$_\varepsilon$ or pdfLATEX is assumed. If not, but `\documentstyle` is defined and `\fmtname` is not `AmS-TeX`, LATEX209 is assumed. If `\pdfoutput` is defined and has a non-zero value, pdfLATEX or pdfTEX is assumed.

4. Files

The current version of MFPIC, consists of five files exclusive of documentation and tests. The files `grafbase.mp` and `grafbase.mf` contain the basic METAPOST and METAFONT macros, the file `dvipsnam.mp` contains the METAPOST definitions of 68 color names. The file `mfpic.tex` contains all the essential TEX macros. The file `mfpic.sty` simply issues `\input mfpic.tex`.

The plain ASCII file `readme.1st` should be (should have been!) read first. The file `manifest.txt` contains the full list of files with an indication of the purpose of each.