MFPIC: A Short Introduction

Daniel H. Luecking*

2010/06/10

Contents

MFPIC version: 1.06.

1 Introduction

As this document aims only to instruct the reader in the building of figures with MFPIC, we will not be too concerned with the intricacies of running programs in various operating systems and TEX distributions. What will be described here is the simplest case: a command-line system in which commands are typed at a keyboard. To simplify things further, we will assume that MFPIC is used with the metapost option, in a LATEX document, with pdfLATEX as the compiler. An appendix will discuss some of the differences when these assumptions are not satisfied.

We will start right out with the "Hello, world" of MFPIC. Construct a LATEX document by typing the following in a text editor and saving it as `first.tex`.

```
% first.tex
\documentclass{article}
  \usepackage[metapost]{mfpic}
  \opengraphsfile{myfigs}
\begin{document}
  My first figure:
  \begin{mfpic}[72]{-1}{1}{-1}{1}
    \ellipse{(0,0),1,.5}
  \end{mfpic}
  \closegraphsfile
\end{document}
```

Run the command

```
pdflatex first
```

which should create several files, the two most important being `first.pdf` and `myfigs.mp`. You can go ahead and open `first.pdf`. You should see a 2 inch by 2 inch square with something similar to '`#1`' in the lower left corner. This shows where the picture will be when it has been created.

Now run the command

```
mpost myfigs
```

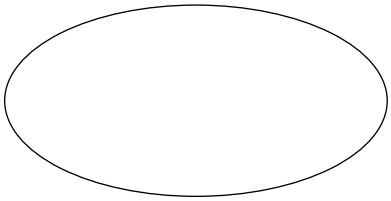
which should create the file `myfigs.1`. This is an EPS file (Encapsulated PostScript) and can be opened in GhostScript or GSview or similar Postscript viewing program to see an ellipse.

If you are viewing `first.pdf` in Acrobat Reader or Adobe Reader, you will need to close it. Now repeat the pdfLATEX step:

```
pdflatex first
```

and then view the file `first.pdf`. You should see something very close to figure 1.1.

What can go wrong? According to Murphy's Law: anything. If MFPIC is not properly installed, one could obtain messages of files not found. If that happens, determine (from your TEX system's documentation) where TEX input files should go and make sure that `mfpic.tex` and `mfpic.sty` reside there. Similarly, find out where METAPOST inputs should go and make sure that `grafbase.mp` and `dvipsnam.mp` reside there. Then run whatever command your TEX system might require to "update the filename database". You may safely ignore the message from MFPIC itself that `myfigs.1` is not found (on the first run of pdfLATEX). This file should be created only after running `mpost`.



My first figure:

Figure 1.1.

If you get an error message from LaTeX, carefully check your typing. Also check whether an older version of MFPIC might have been used instead of the current version. If you get an error message from METAPOST do the same, especially checking the typing within the `mfpic` environment. If you get a message from METAPOST that "Grafbase" believes your MFPIC installation may be broken, check the log files (`first.log` and `myfigs.log`) to find out the locations of these input files:

`mfpic.tex` and `grafbase.mp`

and make sure that both these files are from the most recently installed MFPIC package. If you are only evaluating MFPIC without committing to installing it, just make sure all the files mentioned in the previous paragraphs are in the current directory.

If pdfLaTeX complains it can't write on the file `first.pdf`, unload `first.pdf` from your pdf viewer and try again.

If the figures look a little choppy in Acrobat Reader, turn on "smooth line art" in the edit preferences dialogue.

I will assume that eventually all went well and you are now able to obtain the ellipse of figure 1.1. Each time you change an `mfpic` environment or the options to the package, you potentially change the file `myfigs.mp` produced and you should repeat the sequence:

```
pdflatex first
mpost myfigs
pdflatex first
```

to be sure of seeing the changes.

One thing you might notice about figure 1.1 is that the ellipse is positioned quite a bit above the base line of the text. This is because MFPIC reserves the amount of space specified in the arguments of the `mfpic` environment. These arguments are `[72]{-1}{1}{-1}{1}`, which means that each unit in the picture is 72 times the value of `\mfpicunit`, that is, about one inch. The first pair of mandatory arguments, `{-1}{1}`, indicate the x-coordinates run from -1 to 1. Since these differ by 2, they indicate a width of two inches. The second pair similarly represents a height of two inches. But the ellipse is centered at $(0, 0)$, which is one inch above the bottom (bottom is at $y = -1$), and its vertical radius is .5. So the lowest point on the ellipse should be 0.5 inches above the bottom of the space reserved. MFPIC provides a way to fit the space reserved to the actual extent of 'ink' in the picture. That is by the option truebbox:

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

This would then produce something like figure 1.2. From now on, this option will be in effect in our examples.

My first figure:

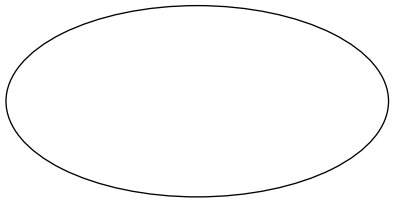


Figure 1.2.

Even though the arguments to the `mfpic` environment are ignored in determining the size of the figure (under `truebbox`), they are still needed in order to establish the coordinate system that the ordered pairs refer to (for example `(0,0)` in the `\ellipse` arguments).

2 Positioning text

By now you are probably thinking: "This so-called 'Hello, world' of MFPIC doesn't say 'Hello, world' anywhere!" We correct that with the following example:

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \ellipse{(0,0),1,.5}
  \tlabel[cc](0,0){Hello, world.}
\end{mfpic}
```

This should give you figure 2.1.

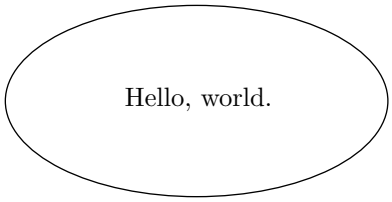


Figure 2.1.

The `\tlabel` command places the given text at the given position $((0,0))$ adjusted according to the optional argument `[cc]`, which says to center the text (both vertically and horizontally) at that location. The `[cc]` is optional. Without it, the text would have the leftmost point of its baseline (the imaginary line that most letters sit on) placed at $(0,0)$.

You are no doubt thinking: "The ellipse doesn't really match the text. What you need is some macro that measures the text and produces an oval with similar dimensions." For that we have the `\tlabeloval` command. The `\tlabeljustify` command in the example below is to communicate to both the text placement and the curve generation procedures that they are to be centered at the point $(0,0)$. (We'll see an easier way to do this later.)

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \tlabeljustify{cc}
  \tlabeloval(0,0){Hello, world.}
\end{mfpic}
```

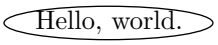


Figure 2.2.

This produces figure 2.2.

This would be better still if a little space is left around the text so the ellipse doesn't touch it. The `\tlpathsep` command can do that:

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \tlabeljustify{cc}
  \tlpathsep{3pt}
  \tlabeloval(0,0){Hello, world.}
\end{mfpic}
```

producing figure 2.3.

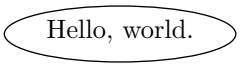


Figure 2.3.

It would be nice to make the text pop out a bit with some color[1]. You can do that by adding `\gfill[yellow]` in front of either `\ellipse` or `\tlabeloval`:

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \tlpathsep{3pt}
  \tlabeljustify{cc}
  \gfill[yellow]\tlabeloval(0,0){Hello, world.}
\end{mfpic}
```

This will produce figure 2.4.

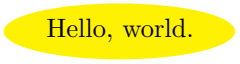


Figure 2.4.

Notice that now the boundary of the oval has not been drawn. This is the standard behavior of MFPIC. A figure command alone will draw the figure. If you want some other rendering than that, you must explicitly provide all of it. To get the boundary back, simply add `\draw` before the `\gfill`. You can draw the curve in a color other than black with an optional argument. We can also make the line thicker with the command `\penwd`:

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \penwd{1.5pt}
  \tlpathsep{3pt}
  \tlabeljustify{cc}
  \draw[blue]\gfill[yellow]\tlabeloval(0,0){Hello, world.}
\end{mfpic}
```

This will produce figure 2.5.

[1] Colors are included in this document only to give examples of their use in MFPIC. I do not necessarily recommend any of them.

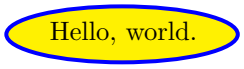


Figure 2.5.

This last version doesn't look too bad, but it seems that the oval ought to be a little fatter (slightly higher than it is now). By default, `\tlabeloval` will make the ratio of width to height the same as that of the text, or rather of the text plus the additional space specified by `\tlpathsep`. This can be changed with an optional argument, a number that multiplies the width-to-height ratio. Decreasing this ratio will decrease the width (slightly) and increase the height. Here we have also omitted the `\tlabeljustify` command and shown that `\tlabeloval` takes a second optional argument that can be used to 'justify' both the curve and the text. To use this, one must explicitly include the first optional argument; if the default is intended, an empty pair of brackets may be used.

```
\begin{mfpic}[72]{-1}{1}{-1}{1}
  \penwd{1.5pt}
  \tlpathsep{3pt}
  \draw[blue]\gfill[yellow]\tlabeloval[.8][cc](0,0){Hello, world.}
\end{mfpic}
```

This will produce figure 2.6.

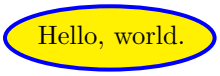


Figure 2.6.

The `\tlabeloval` command places the label last, after the action of all the preceding macros; therefore the text ends up on top of everything else. The `\tlabeloval` command also has a '*-form' that does everything *except* place the text. Finally, ovals are not the only thing that can be used to surround text. See the manual (`mfpic-doc.pdf`) and below for others.

Here is a more common use of `tlabel` commands: labeling a graph and axes. In the following example we have given `\tlabel` the option `[bl]` to place the bottom left corner of the text at the given coordinates. However, we have used `\tlpointsep{3pt}`, which has the effect of shifting text away from its nominal location, to prevent the text from colliding with the curve.[2] The value set by `\tlpointsep` has an effect only if the point is on the edge of the text, so there would be no shifting with the `[cc]` placement used earlier.

```
\begin{mfpic}[72]{0}{2.5}{0}{1}
  \tlpointsep{3pt}
  \polyline{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
  \tlabel[bl](.5,1){Max output}
  \dashed\polyline{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
  \tlabel[bl](1.5,.7){Max input}
\end{mfpic}
```

This will produce figure 2.7.

[2] One can also use `\tlabelsep`, which is equivalent to `\tlpathsep` plus `\tlpointsep`.

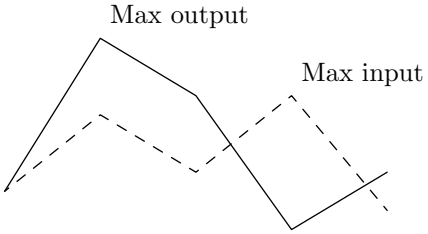


Figure 2.7.

Notice that `\polyline` alone produces a solid line while `\dashed\polyline` makes a dashed line. Let us close this section by dressing up this figure with axes, some fat dots marking the keypoints, and hash marks on the axes:

```
\begin{mfpic}[72]{0}{2.5}{0}{1}
  \tlpointsep{3pt}
  \polyline{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
  \point[3pt]{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
  \tlabel[bl](.5,1){Max output}
  \dashed\polyline{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
  \pointfillfalse
  \point[3pt]{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
  \tlabel[bl](1.5,.7){Max input}
  \axes
  \xmarks{0,0.5,1,1.5,2}
  \axislabels x{{$50$} .5, {$100$} 1, {$150$} 1.5, {$200$} 2}
\end{mfpic}
```

This will produce figure 2.8.

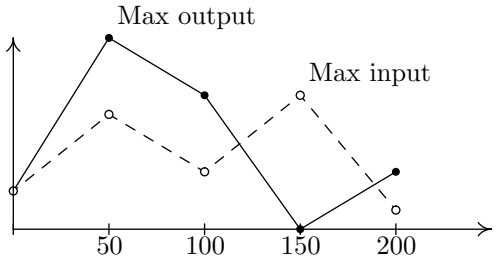


Figure 2.8.

The optional argument of `\point` specifies the diameter of the points to draw. The command `\pointfillfalse` forces the points to be drawn as open circles. The axes configure themselves to the size specified in the argument of the `mfpic` environment. The `axislabels` command takes as arguments a letter, to specify the axis, and a comma separated list of labels, each of which is specified by some text to place (in braces) and the x-coordinate to place it at.

3 Drawing figures

MFPIC has several predefined figures and commands to obtain essentially any curve (provided one can obtain enough points on it with sufficient precision). We've already seen `\polyline` and `\ellipse`. The former needs a list of points to connect with line segments and the latter needs the center and radii of the ellipse. The `\ellipse` also takes an optional argument: the number of degrees to rotate the ellipse. Here we list some of the more common such figures. Remember that all of them will produce some sort of line drawing if used alone. They can be preceded by `\dashed` to make the lines dashed or `dotted` to make them dotted. If the figure is a closed curve, `\gfill` will fill them in.

```
\begin{mfpic}[72]{0}{4}{0}{1}
  \rect{(0,0),(1,.75)}
  \circle{(1.5,.5),.45}
  \arc[s]{(3,0),(2,1),45}
  \ellipse[20]{(3.5, 0.5), 0.6, 0.4}
\end{mfpic}
```

This produces figure 3.1. The `\arc` command has several forms. The optional argument picks the form to use. This one specifies the endpoints of the circular arc and the angle of the arc (the angle between the radii from the center of the circle to those two points). Other possibilities are a three-point form (option `[t]`), a polar form (option `[p]`), and a center-point-sweep form (option `[c]`, specify a center, starting point, and angle). See the manual for details. The default (what would be assumed if no optional argument is given) is `[s]` and is called the point-sweep form.

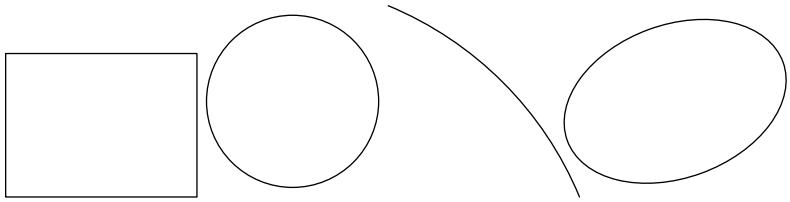


Figure 3.1.

The `\polyline` command draws straight lines connecting points. We can also draw smooth curves. Lets take the same points from our `\polyline` example (figure 2.7), but change `\polyline` to `\curve`, omit the text, and add the points from figure 2.8:

```
\begin{mfpic}[72]{0}{2.5}{0}{1}
  \curve{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
    \point[3pt]{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
  \dashed\curve{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
    \pointfillfalse
    \point[3pt]{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
\end{mfpic}
```

This should produce figure 3.2.

This is somewhat unsatisfying. One could improve the result by selecting more points, or by increasing the 'tension' in the curve.

Roughly speaking, tension determines how straight the segments between the points are, and how sharp the turns at each point. High tension makes the curve look a little

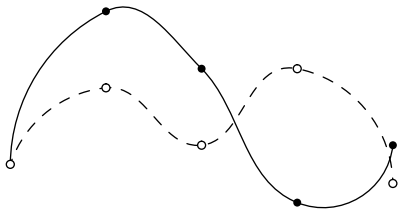


Figure 3.2.

more like a polyline. The default tension is 1, a tension of about 5 makes the result look somewhat like a polyline with very slightly rounded corners, very high tensions make the curve indistinguishable from a polyline. Tension must (almost) always be greater than 0.75.

Another effect of increased tension is to reduce the little wobbles we can see in the first curve. Let's try a tension of 1.5, which can be specified as an optional argument to `\curve`:

```
\begin{mfpic}[72]{0}{2.5}{0}{1}
  \curve[1.5]{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
    \point[3pt]{(0,.2),(.5,1),(1,.7),(1.5,0),(2,.3)}
  \dashed\curve[1.5]{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
    \pointfillfalse
    \point[3pt]{(0,.2),(.5,.6),(1,.3),(1.5,.7),(2,.1)}
\end{mfpic}
```

This give figure 3.3.

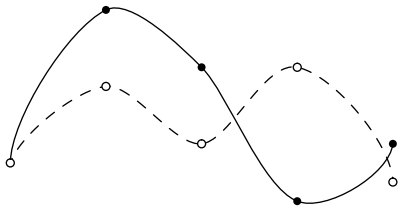


Figure 3.3.

When we use `\curve`, there is no way METAPOST can tell if we are just connecting points or if we are trying to graph a function. It *cannot* enforce the requirement, which every function must satisfy, that the curve should travel left-to-right. The command `\fcncurve` does enforce this (assuming the points to be connected are listed in left-to-right order). This command also permits an optional tension argument. The dotted line in figure 3.4 is produced with `\curve`, the solid one with `\fcncurve`. One might conceivably want to decrease the tension a bit here.

```
\begin{mfpic}[72]{0}{2.5}{0}{1}
  \dotted\curve{(0,.2),(.5,0),(.85,.5),(1,1),(1.5,0),(2,.3)}
  \fcncurve{(0,.2),(.5,0),(.85,.5),(1,1),(1.5,0),(2,.3)}
    \pointfillfalse
    \point[3pt]{(0,.2),(.5,0),(.85,.5),(1,1),(1.5,0),(2,.3)}
\end{mfpic}
```

Other figures available include

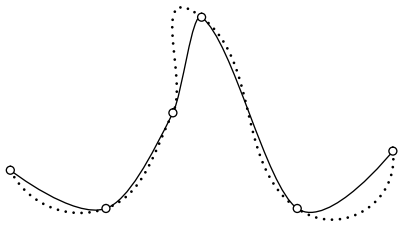


Figure 3.4.

`\cyclic` Used just like `\curve` but closes the path (connects the last point smoothly to the starting point).

`\polygon` Used just like `\polyline` except it connects the last point to the first with a straight line.

`\sector` Makes a wedge with two straight lines and an arc. The arguments are almost the same as `\arc[p]`, but the order is different: center, radius and two angles.

Here are some other curves that, like `\tlabeloval`, are proportioned to fit given text. All have a `*`-form that draws the path without placing the text.

`\tlabelrect` This produces a rectangle. It has the same usage as `\tlabeloval`, except the first optional argument specifies the radius of quarter-circles used to make rounded corners.

`\tlabelellipse` This is similar to `\tlabeloval` except that instead of modifying the width-to-height ratio, the first optional argument *is* the width-to-height ratio. If that argument is 1 (the default) you get a circle.

`\tlabelcircle` This produces a circle, of course.

4 Functions

METAPOST is able to calculate a number of functions natively, and still more have been defined in MFPIC. Also available are the usual arithmetic operations. Any valid METAPOST expression, containing one unknown `x` and producing a numerical result can be graphed.

Here is an example of the graphs of $y = x^2$ and $y = \pm\sqrt{x}$. Note that exponentials are denoted by `**` and it is important to note that it has the same precedence as multiplication (denoted by a single `*`). That is, in a formula like `3*3**2`, the operations are performed in order, left to right, producing $(3 \cdot 3)^2 = 81$ and not $3 \cdot 3^2 = 27$. Parentheses are needed if the latter is intended: `3*(3**2)`.

```
\setlength{\mfpicunit}{1cm}
\begin{mfpic}{-2.5}{2.5}{-1.5}{4}
  \function{-2,2,.1}{x**2}
  \function{0,2,.1}{sqrt x}
  \function{0,2,.1}{-sqrt x}
  \axes
  \xmarks{-2,-1,1,2}
  \ymarks{-1,1,2,3}
  \tlpointsep{3pt}
  \axislabels x{{$-2$}-2,{$-1$}-1,{$1$}1,{$2$}2}
  \axislabels y{{$-1$}-1,{$1$}1,{$2$}2,{$3$}3}
\end{mfpic}
```

This produces figure 4.1.

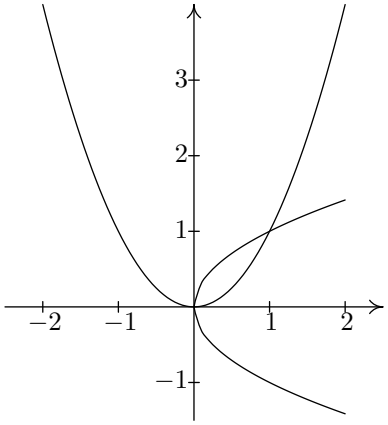


Figure 4.1.

The command `\function` has two arguments. The first contains the starting and ending x-values of the desired graph, followed by a *step size*. Generally the smaller the steps the better the accuracy, but METAPOST has a limit on the number of steps (usually about 2000). There is also an optional argument which can be `[s]`, the default, which means the graph is to be smooth, or `[p]`, which means the graph is constructed by connecting the calculated points with straight lines. Here is the same example with larger step size to emphasize the difference (see figure 4.2)

```
\setlength{\mfpicunit}{1cm}
\begin{mfpic}{-2.5}{2.5}{-1.5}{4}
  \function[p]{-2,2,.5}{x**2}
  \function[p]{0,2,.5}{sqrt x}
  \function[p]{0,2,.5}{-sqrt x}
  \axes
  \xmarks{-2,-1,1,2}
  \ymarks{-1,1,2,3}
  \tlpointsep{3pt}
  \axislabels x{{$-2$}-2,{$-1$}-1,{$1$}1,{$2$}2}
  \axislabels y{{$-1$}-1,{$1$}1,{$2$}2,{$3$}3}
\end{mfpic}
```

In addition, one can increase the tension in the curve drawn by putting a tension value after the `s` in `[s]`. For a tension of 2.4: `\function[s2.4]{...}`.

The functions available include `sqrt` and all the trig functions: `sin x` assumes `x` is an angle in radians, `sind x` assumes it is in degrees, with a similar naming convention for the remaining trig functions. The inverses are `asin x`, `acos x`, and `atan x`, which produce angles in degrees, and `invsin x`, etc., which produce angles in radians. There is also `ln x` or `log x` for the natural logarithm, `exp x` for e^x, `logten x` for the base 10 logarithm, `logtwo x` for base 2, and `logbase` for other bases: `logbase(16) x` (for example) for base 16. The general syntax of these functions is the following: if the argument is `x` alone or a pure number alone or the particular case of a number followed by `x` (no `*` in between!) then

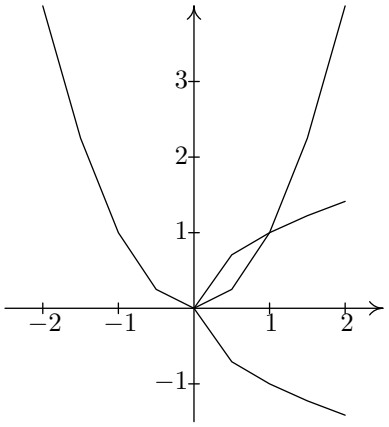


Figure 4.2.

parentheses are not needed. Example: `sin 2x`. For almost anything else, parentheses are required: `sin(3*x)` or `sin(x**2)`.

Some other functions available are the hyperbolic functions, `sinh x`, `cosh x`, etc. (all 6 of them), and the inverses of three of them: `asinh x`, `acosh x`, and `atanh x`.

These functions (or any METAPOST numeric expression) can also be used in any of the coordinates of points in drawing commands like `\polyline` (but not usually in text placement commands like `\tlabeloval`). For example (from now on the value of `\mfpicunit` is set to `1cm`):

```
\begin{mfpic}{-.5}{2.5}{-1.5}{1.5}
  \polyline{(2,-sqrt 2),(1,-1),(.5,- sqrt .5),(0,0),
      (.5,sqrt .5),(1,1),(2,sqrt 2)}
  \axes
  \xmarks{1,2}
  \ymarks{-1,1}
  \tlpointsep{3pt}
  \axislabels x{{$1$}1,{$2$}2}
  \axislabels y{{$-1$}-1,{$1$}1}
\end{mfpic}
```

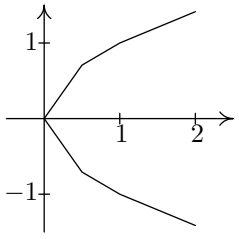


Figure 4.3.

There are other types of functions: parametric functions, and polar coordinate versions. MFPIC provides `\parafcn` and `\plrfcn` to graph these. The `\parafcn` requires a starting

value, and ending value and a step size just as in `\function`, but in the second argument there must be either a pair of expressions in the variable `t`, separated by a comma and enclosed in parentheses, or a single *pair-valued* expression. METAPOST and MFPIC provide only a few pair-valued functions; one is used below.

The second argument of `\plrfcn` must contain a single numeric expression in the variable `t`, and indicates a function of θ to be graphed in polar coordinates: $r = f(\theta)$. In the following example (figure 4.4), we draw a portion of the graph of $x = y^2$ by representing it as the graph of the parametric equations $x = t^2$, $y = t$, and a portion of a circle of radius 1.5 by representing it as the graph of the pair-valued function `dir(t)`. The expression `dir(t)` gives the point whose distance from $(0, 0)$ is 1 in the direction given by the angle `t`.

```
\begin{mfpic}{-2}{4}{-2}{2}
  \parafcn{-2,2,.1}{(t**2,t)}
  \dotted\parafcn{45,315,5}{1.5*dir(t)}
\end{mfpic}
```

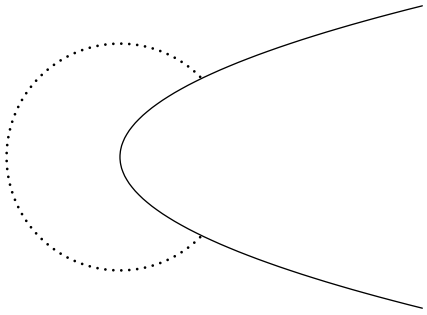


Figure 4.4.

Here is an example of a graph of the polar coordinate function $r = 2 \sin 3\theta$ (figure 4.5). We use the degree version `sind` in order to work with integers.

```
\begin{mfpic}{-2}{2}{-2}{2}
  \plrfcn{0,180,5}{2*sind 3t}
\end{mfpic}
```

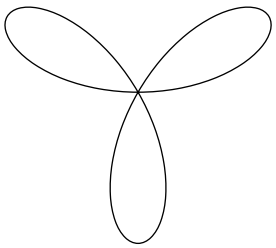


Figure 4.5.

5 Transforming figures

METAPOST is capable of any affine transformation (things like shifting, rotating, scaling, reflecting and slanting) of any path. The figures we've been dealing with so far (`\ellipse`,

`\curve`, `\function`, etc.) all produce, in the METAPOST code, the definition of some path (as well as a drawing of that path). MFPIC provides for different methods of 'drawing' the path with *prefix macros*. We've seen `\dashed`, `\dotted`, `\gfill` so far, in addition to the default `\draw`. MFPIC also provides for modifying the shape and position of the path with other prefixes. Here's a simple example.

```
\begin{mfpic}{-.5}{2.5}{-.5}{2.5}
  \rotatepath{(1,.5), 45}\rect{(0,0),(2,1)}
  \point{(1,.5)}
\end{mfpic}
```

The command `\rotatepath` obviously rotates the path that follows, but it needs to know what the center of rotation will be, and how much to rotate. These are given in its mandatory argument, separated by a comma. The example above (pictured in figure 5.1) rotates 45 degrees around the center of the rectangle.

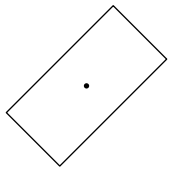


Figure 5.1.

Notice that we have no drawing prefix. A combination of transformation-plus-figure is treated as a figure in its own right and behaves the same. If we want the figure dashed, we could write

```
\dashed\rotatepath{(1,.5),45}\rect{(0,0),(2,1)}
```

It may not be obvious, but we can also write a drawing macro between the rotation and the figure, producing figure 5.2

```
\begin{mfpic}{-.5}{2.5}{-.5}{2.5}
  \rotatepath{(1,.5), 45}\draw\rect{(0,0),(2,1)}
  \point{(1,.5)}
\end{mfpic}
```

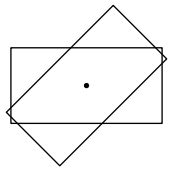


Figure 5.2.

This illustrates another property of MFPIC macros: the combination of a rendering prefix and a figure is also treated the same as a figure in its own right: the same figure as the one that follows. In fact, the only difference between `\rect` and `\draw\rect` in this example is that the second one has a minor (!) side effect: the rectangle is drawn.

Finally, try to guess what happens if we add another prefix at the front:

```
\begin{mfpic}{-.5}{2.5}{-.5}{2.5}
  \dotted\rotatepath{(1,.5), 45}
    \draw\rect{(0,0),(2,1)}
\end{mfpic}
```

and if we add another rotation in front of that.

```
\begin{mfpic}{-.5}{2.5}{-.5}{2.5}
  \rotatepath{(0,0),45}
    \dotted\rotatepath{(1,.5), 45}
      \draw\rect{(0,0),(2,1)}
\end{mfpic}
```

Available transformations include

`\scalepath`, `\shiftpath`, `\xscalepath`, `\yscalepath`, `\slantpath`, and `\reflectpath`.

See the manual for a description of what arguments are required for each. Here's a final example, producing figure 5.3

```
\begin{mfpic}{-.5}{2.5}{-.5}{2.5}
  \shiftpath{(-1,1)}\draw[red]\slantpath{.5,1}\dotted
  \rotatepath{(0,0), 90}\dashed\rect{(0,0),(2,1)}
  \point{(0,0),(2,1)}
  \tlpointsep{2pt}
  \tlabel[tr](0,0){$(0,0)$}
  \tlabel[bl](2,1){$(2,1)$}
\end{mfpic}
```

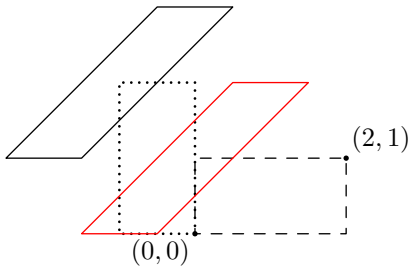


Figure 5.3.

6 Rendering figures

Rendering is the act of making a description of a figure visible. Examples are: drawing a solid curve, drawing a dashed curve, or filling its interior, For MFPIC figure macros the default, in the absence of explicit commands, is to use `\draw`. That is,

```
\rect{(0,0),(1,2)}
```

has the same result as

```
\draw\rect{(0,0),(1,2)}
```

The default rendering can be changed. Just say `\setrender{\dashed}`, and all figures afterward will be dashed (see figure 6.1).

```
\begin{mfpic}{0}{2}{0}{1}
\setrender{\dashed}
  \rect{(0,0),(1,1)}
  \circle{(1.5,.5),.5}
\end{mfpic}
```

Figure 6.1.

The `\setrender` command can be inside an `mfpic` environment to affect only later commands in that figure, or outside to affect all later MFPIC figures.

We give a few examples now of the renderings possible. These divide more-or-less into those that trace a path and those that fill in a path. In order to fill in a path, it must be a closed path, of course, but METAPOST distinguishes between closed paths and those that merely happen to end where they began. There is a good reason for this: METAPOST cannot, without human aid, know if two points are the same, or merely accidentally so close that the accuracy of the program sees them as the same. It requires human aid in the form of an explicit request to create a closed path. Of the MFPIC macros we've seen so far, `\ellipse`, `\circle`, `\rect`, `\polygon`, and `\cyclic` produce closed paths, but `\polyline`, `\curve`, `\function`, `\parafcn`, and `\plrfcn` do not. Also producing closed paths are `\tlabeloval` and its relatives.

The following example illustrates filling with a hatching pattern (parallel lines) and an *unfilling*. Clearing the interior of a path may not seem like rendering, but it is treated in exactly the same way (think of it as a negative rendering). We first hatch a rectangle, then clear out a smaller rectangle with rounded corners to place our text inside. The results are in figure 6.2.

```
\begin{mfpic}{0}{2}{0}{2}
  \draw[red]\lhatch[2pt][blue]\rect{(0,0),(2,2)}
  \gclear\tlabelrect[6pt][cc](1,1){Hatching!}
\end{mfpic}
```

Figure 6.2.

This example illustrates that `\lhatch` fills with left slanting lines. And that it takes two optional arguments. The first is the distance between lines, and the second is the color to make the lines. There are also `\rhatch` which slants the lines the other way, `\xhatch` which uses both slants, and `\thatch` which can draw the lines at any angle.

Here is another example of rendering (figure 6.3). The new macro is `\polkadot`. We've repeated this example twice to show the effect of changing the order of the prefixes. Each prefix applies its rendering to the result of everything to the right of it. In the second example

the hatching goes over the dots (and a bit of the dashes as well). If the \gfill were first, it would cover almost everything else.

```
\begin{mfpic}{0}{6}{0}{2}
  \penwd{2pt}
  \hatchwd{2pt}
  \drawcolor{blue}
  \hatchcolor{red}
  \fillcolor{green}
  \dashed\polkadot\rhatch[5pt]\gfill[yellow]\rect{(0,0),(2.8,1.8)}
  \rhatch[5pt]\dashed\polkadot\gfill[yellow]\rect{(3,0),(5.8,1.8)}
\end{mfpic}
```

We've added a couple of other new features to this example. To emphasize effects, we've increased the thickness of the drawing pen (\penwd) and the hatch lines (\hatchwd). We've also used the \drawcolor macro and its relatives to set the colors to be used. The \polkadot macro uses the color set by \fillcolor; so does \gfill if no optional color is given.

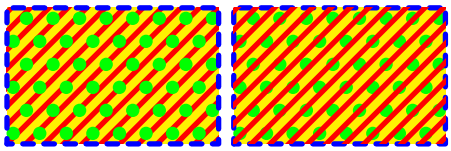


Figure 6.3.

If one wants to plot several curves in a single graph, they often need to be rendered differently. The three methods we've seen so far, \draw, \dashed, and \dotted, may not be enough. The \dashed and \dotted commands permit an optional argument to adjust the length of the dashes and spaces, and size of the dots. One can also change the curve thickness with \penwd. But that may not be 'different' enough. MFPIC provides a few solutions. When color is available, they may be drawn in different colors. When not, there are two possibilities: \gendashed and \plot.

The first, \gendashed, is a generalized dashing macro. It takes one mandatory argument, the name of a dashing pattern. Named dashing patterns may be created with the \dashpattern command, as shown by the following example (see figure 6.4):

```
\begin{mfpic}{-3.5}{3.5}{-1.2}{1.2}
  \dashpattern{dotdash}{0pt,4pt,3pt,4pt}
  \gendashed{dotdash}\function{-pi,pi,.2}{sin 2x}
  \function{-pi,pi,.2}{cos 2x}
  \axes
\end{mfpic}
```

The \dashpattern command takes a name and an even number of lengths. The first, third, etc., lengths represent the lengths of dashes (0pt means a dot), and the second, fourth, etc., represent spaces. The given pattern is dot-space-dash-space. This pattern, when used in a \gendashed command, is repeated for the length of the curve.

This last example illustrates that the predefined METAPOST variable pi (equal to 3.14159) can be used pretty much anywhere a number can be used (except, often, in text label commands).

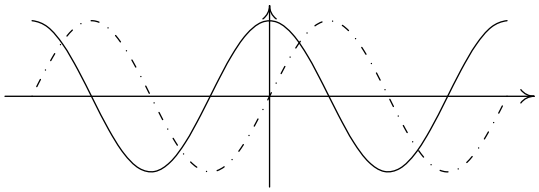


Figure 6.4.

Another way to get more distinctive curves is to 'dot' them with something other than tiny dots. The `\plot` command does that. It takes one mandatory argument, the name of a symbol to use instead of a dot. Here are the same two curves `\plot`-ed (figure 6.5):

```
\begin{mfpic}{-3.5}{3.5}{-1.2}{1.2}
  \setlength{\pointsize}{2.5pt}
  \plot{Triangle}\function{-pi,pi,.2}{sin 2x}
  \plot[2pt,6pt]{SolidCircle}\function{-pi,pi,.2}{cos 2x}
  \axes
\end{mfpic}
```

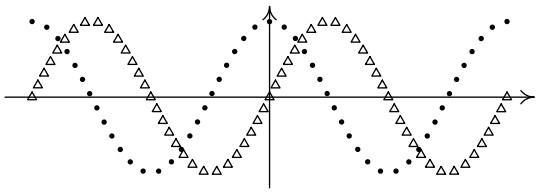


Figure 6.5.

The `\plot` command takes an optional argument to specify the size of the symbols and the spacing between them. The size of the symbols can also be adjusted by changing the length command `\pointsize` (that also adjusts the size of the dots placed with the `\point` command).

In this last example, `\plotnodes` is similar to `\plot`, except it placed the symbols at the 'nodes' defined by the path command. In the case of `\function`, these are the points $(x_k, f(x_k))$ with x_k stepping through all the x-values determined by the first argument of `\function` (figure 6.6).

```
\begin{mfpic}{-3.5}{3.5}{-1.2}{1.2}
  \plotnodes[2.5pt]{Square}\function{-pi,pi,pi/16}{sin 2x}
  \axes
\end{mfpic}
```

See the manual for the list of predefined symbols available to the `\plot` and `\plotnodes` command.

7 More on text

The text positioning commands used so far in this guide are entirely handled by TEX or LATEX. This is why we have occasionally had to say that certain things could be done "except in text placement commands". It is possible for text positioning to be done within

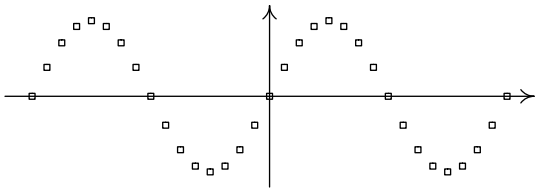


Figure 6.6.

METAPOST, making many things possible that couldn't be done otherwise. For example, text can be rotated about the point of placement. You are probably thinking that LaTeX can rotate text, but it is not all that easy to arrange for the point on the graph where we place the text to be the center of rotation. Below are two examples, in which we attempt to place the text separated from $(0,0)$ by `5pt` and rotated 45 degrees around $(0,0)$. In the first we try to use LaTeX's `\rotatebox` command, and in the second we turn on METAPOST handling of labels and use a rotation option to the `\tlabel` command.

```
\begin{mfpic}{0}{1}{0}{1}
  \point{(0,0)}
  \polyline{(0,0),(1,1)}
\tlabel[Bl](0,0){\rotatebox{45}{\hspace{5pt}Test text}}
\end{mfpic}

\usemplabels
\begin{mfpic}{0}{1}{0}{1}
  \point{(0,0)}
  \polyline{(0,0),(1,1)}
  \tlpointsep{5pt}
  \tlabel[Bl45](0,0){Test text}
\end{mfpic}
```

(a) (b)

Figure 7.1.

The first produces figure 7.1a and the second produces figure 7.1b. Our goal was to get the baseline of the text lined up with the reference line drawn.

In the first example, LaTeX's `\rotatebox` command produces the following result, where we put a frame around both the unrotated text and the rotated result to emphasize what LaTeX sees as the boundaries:

This is then placed by the `\tlabel` command with the lower left corner of the *outer* box at $(0,0)$. But LaTeX's axis of rotation was at the lower left corner of the inner box. In

the second case, METAPOST placed the label. The command `\tlpointsep{5pt}` and the parameter `[Bl45]` explicitly request that the label be placed with its left baseline 5 points from $(0,0)$ and rotated 45 degrees *about the point* $(0,0)$.

The `\usemplabels` command used above asks METAPOST to arrange for the setting of labels. Adding the option mplabels to the `\usepackage` command that loads MFPIC has the same effect for the whole document. There can be problems with using METAPOST to set labels. One is that METAPOST has to call a `tex` program to do the actual typesetting, and then one must either make arrangements that ensure METAPOST will call LaTeX, or never use any macros in the labels that are not defined in plainTeX. If one does arrange for LaTeX to be used, one needs to arrange that a LaTeX preamble is prepended to the output `.mp` file. The `\mfpverbtex` command can be used for this.

The command `\nomplabels` can be used to return to having labels set at the document level. For the rest of this guide, we have mplabels in effect.

There are a few more commands that place text on the picture. All of them pass the final responsability for text placement to METAPOST if mplabels is in effect. See the manual for more details.

8 Arrows

The command `\arrow` adds an arrowhead onto the *end* of any path that follows. For this to have predictable effects, you need to know which part of a curve is the end, and which the start. Not surprisingly, for the commands that connect a list of points in order the first point in the list is the start point and the last point is the end. Except the closed paths (`\cyclic`, `\polygon`, etc.); for them, the start and the end points are the same, but the order of the points gives a direction to the arrowhead. The default `\circle` has an anticlockwise direction, but if the circle is defined by three points (for example) the direction of the circle is determined by the order in which the points are written.

Anyway, here are a few examples, illustrating the use of `\arrow`, and some of its optional arguments.

```
\begin{mfpic}{0}{4}{0}{4}
  \arrow[r5]\circle{(1,1),.5}
  \arrow[b4pt]\arrow\polyline{(3,2),(3,0)}
  \arrow[cred]\reverse\arrow\polyline{(0,3),(2,3)}
  \arrow[l 5pt]\rect{(4,2),(2,4)}
\end{mfpic}
```

See figure 8.1 for the results of this example. There are four possible optional arguments, the first character inside the brackets tells what option the rest of the argument applies to. The first example above starts with '`r`', which stands for 'rotate' and asks for the arrowhead to be rotated 5 degrees (positive rotation means anticlockwise, negative means clockwise). This is frequently useful for arrows on curved paths, as the default direction (tangent to the path) often just looks wrong). The second example starts with '`b`', which stands for 'backset' and it moves the head back `4pt` from where it would otherwise be placed. In the example, this is used to put a double arrowhead on the line. In the third example we put an arrow at both ends by reversing the sense of the curve in between the two `\arrow` prefixes. We also used the letter '`c`' in the optional argument of one arrowhead. This stands for 'color' and the requested color is '`red`'. Finally, the `l` option (that's a lowercase 'ell', not the number 'one') changes the length of the arrowhead to 5 points (from the `3pt` default).[3]

[3] I have put a space between the `l` and the `5pt` so it won't be mistaken for '15pt. Normally one should

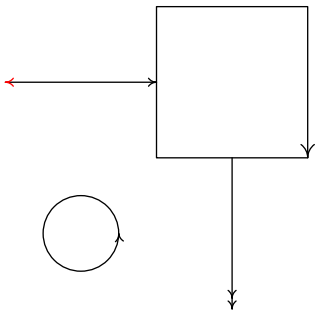


Figure 8.1.

The options can be combined in one command: `\arrow[cblue][b4pt][r25][l6pt]` would produce a `6pt` long blue arrowhead rotated 25 degrees anticlockwise, set back `4pt`. The setting back is done in the direction determined *after* rotation. The order of the options is not significant.

The shape of the arrowhead can be changed with the `\headshape` command. The following example draws the arrowhead first normally, and then after an instance of this command. We draw it a third time, exactly like the second time, except we use the *-form. We have increased the length of head and the thickness of the pen to emphasize the effects.

```
\begin{mfpic}{0}{4}{0}{4}
  \setlength{\headlen}{20pt}
  \penwd{3pt}
  \arrow\polyline{(0,3),(4,3)}
    \headshape{.5}{2}{true}
  \arrow\polyline{(0,2),(4,2)}
  \arrow*\polyline{(0,1),(4,1)}
\end{mfpic}
```

The results are pictured in figure 8.2. The first argument to `\headshape` sets the ratio of width to height for the head. We have cut it in half here. The second argument sets the tension in the curves that form the sides of the head. This reduces the curvature in the sides. The third argument can be only `true` or `false` and determines whether the head is a solid shape, or only the two 'barbs'. The defaults correspond to `\headshape{1}{1}{false}`. The filled form does not draw the outline so what we see is the pointy arrowhead on top of a thick line. The *-form tries to erase part of the line so that one sees an actual pointy arrow.

9 Color

We saw the use of color in earlier sections, and now it's time to be systematic about it. The several rendering commands have a color option; examples are `\draw`, `\gfill`, `\arrow`, and the hatching commands. However, even those commands that don't provide such an option can have the color of their rendering changed. MFPIC provides the following commands to change certain colors. Those commands with a color option can be used without that option and then they will the use the appropriate color described here. Each of these color-changing

avoid spaces in MFPIC optional arguments, but this is one case where it will cause no harm.

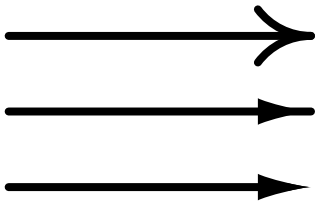


Figure 8.2.

commands takes a mandatory argument containing the color to change to, and an optional argument to be described later.

`\backgroundcolor` This sets the color to be used by `\gclear`. It is the same color used by `\point` for the inside of the points when `\pointfillfalse` has been used. In METAPOST, the only way to clear the inside of a region is to cover it up. The default color for this purpose is `white`. Use this command to change that default.

`\drawcolor` This sets the default color used by those rendering commands that draw a path. This includes `\draw`, but also includes `\dashed`, `\dotted`, `\plot` and `\plotnodes`. It is also used by other commands that produce lines or curves: figure macros used without any rendering prefix, as well as `\axes` and related commands.

`\fillcolor` This sets the default color used by `\gfill`. It is also used by `\polkadot` (which has no color option).

`\hatchcolor` This sets the default color used by any hatching command.

`\headcolor` This sets the default color for arrowheads added by the `\arrow` command. It is also the color of arrowheads on any coordinate axis.

`\pointcolor` This sets the color used by `\point`, `\grid`, and `\plotsymbol` (the last one will be described later).

`\tlabelcolor` This sets the color used for all text labels if the mplabels option is turned on.

The color can be a common name for a color, provided that name is one of the following: `white`, `black`, `red`, `green`, `blue`, `cyan`, `magenta`, or `yellow`. We have already seen this usage. It can also be a color name defined in the file `dvipsnam.mp` that accompanies MFPIC. It can also be an explicit color formula, where color formulas are described in the MFPIC manual.

The optional argument is one of the *color models*. See the manual for details, but the syntax is just like that of the COLOR package's `\color` command. For example,

```
\pointcolor[rgb]{0,1,0}
```

would use the color model `rgb` with parameters 0, 1, and 0 (this is green). After each of these commands a certain color name is assigned a value. For example, a use of the `\pointcolor` command assigns a value to the color named `pointcolor`. Also `\drawcolor` sets `drawcolor` and this pattern is followed for all the color setting commands above except `\backgroundcolor`, which assigns its value to the color named `background`.

Color names for MFPIC use can be defined using the `\mfpdefinecolor` command. Here's an example (figure 9.1). Note the use of the color name `pointcolor` to make arrowheads and points have the same color.

```
\begin{mfpic}{0}{3.5}{0}{3.5}
  \tlabelcolor{red}
```

```
  \pointcolor{rgb(0,1,0)}% green
  \drawcolor[rgb]{0,0,1} % blue
  \fillcolor{Goldenrod}  % from dvipsnam.mp
  \headcolor{pointcolor} % will be green after above
  \mfpdefinecolor{DarkerRed}{rgb}{.67,0,0}
  \hatchcolor{DarkerRed}
  \penwd{1pt}
  \gfill\circle{(1,1),.5}
  \point[3pt]{(1,.5),(1,1.5),(.5,1),(1.5,1)}
  \hatch\rect{(2.5,2.5),(3.5,3.5)}
  \arrow[l 5pt]\polyline{(1,1),(3,3)}
  \tlabel[cc](1,3){Examples\\of\\colors}
\end{mfpic}
```

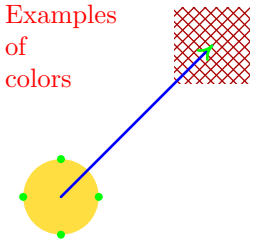


Figure 9.1.

10 Closing paths

There are many different ways to modify a figure. We have already seen `\arrow`, which appends an arrowhead, `\reverse` which reverses the sense, and several that apply an affine transformation (`\rotatepath`, `\shiftpath`, etc.). Now we will see the simple operation of closing a path.

All methods of closing a path have to connect the end to the start, but simply drawing a connection is not enough. METAPOST has to be told to close the path, and what kind of connection is desired. We have several macros that can do the job, the simplest being `\lclosed`, which closes with a straight line. Putting `\lclosed` in front of `\polyline`, for example, produces the same result as `\polygon`. Another macro is `\sclosed` which produces a smooth closure. Putting it in front of `\curve` gives (almost) the same result as `\cyclic`. There is one other useful macro, `\bclosed`, which also informs METAPOST to make a smooth closure. The difference between `\sclosed` and `\bclosed` is that the first modifies slightly the original path (in order to achieve the effect that `\sclosed` + `\curve` ≈ `\cyclic`), the second just asks METAPOST to do its best to connect the ends smoothly. Here's an example comparing the two smooth methods (figure 10.1).

```
\begin{mfpic}{0}{4}{0}{4}
 % an open curve:
  \curve{(0.49,3),(.5,3.7),(1,4),(1.5,3.7),(1.51,3)}
 % \sclosed a shifted copy:
  \draw\gfill[green]\sclosed\shiftpath{(2,0)}
      \curve{(0.49,3),(.5,3.7),(1,4),(1.5,3.7),(1.51,3)}
```

```
 % \bclosed another copy:
  \draw\gfill[yellow]\bclosed\shiftpath{(2,-2)}
      \curve{(0.49,3),(.5,3.7),(1,4),(1.5,3.7),(1.51,3)}
 % \cyclic with same points, shifted:
  \draw\gfill[red]\shiftpath{(0,-2)}
      \cyclic{(0.49,3),(.5,3.7),(1,4),(1.5,3.7),(1.51,3)}
  \tlabeljustify{bc}
  \nomplabels
  \tlabels{
    (1,2.4){\cs{curve}}
    (3,2.4){\cs{sclosed}}
    (1,0.4){\cs{cyclic}}
    (3,0.4){\cs{bclosed}}
  }
 % Some points to help illustrate
  \point{(0.49,3),(.5,3.7),(1,4),(1.5,3.7),(1.51,3)}
  \point{(2.49,3),(2.5,3.7),(3,4),(3.5,3.7),(3.51,3)}
  \point{(0.49,1),(.5,1.7),(1,2),(1.5,1.7),(1.51,1)}
  \point{(2.49,1),(2.5,1.7),(3,2),(3.5,1.7),(3.51,1)}
\end{mfpic}
```

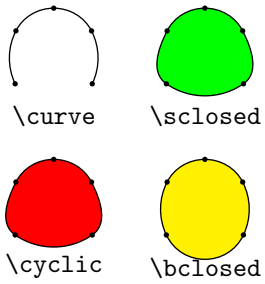


Figure 10.1.

A word about the labels: we turned off mplabels with the command `\nomplabels`, because we used a command (`\cs`) defined for this document and not known to basic TEX or LATEX. The labels therefore are position by LATEX while it assembles this document, instead of by METAPOST which would call a separate instance of TEX or LATEX where `\cs` was unknown. We could have kept mplabels, provided we had used `\mfpverbtex` to write the appropriate LATEX preamble to the `.mp` output. It would need to be some subset of the preamble of this document.

11 Appendices

In addition to pdfLATEX, MFPIC works with plain pdfTEX, LATEX, and plain TEX. Instead of METAPOST as the figure processor, METAFONT can also be used. Let's start with the difference between using MFPIC in a plain TEX document and using it in a LATEX document.

11.1 MFPIC in plain TEX

Here is a sample plain pdfTEX document with results the same as our first "Hello, world" example. Let's call this file `plfirst`

```
\input mfpic
  \usemetapost
\opengraphsfile{myfigs}
  My first figure:
  \mfpic[72]{-1}{1}{-1}{1}
    \ellipse{(0,0),1,.5}
  \endmfpic
  \closegraphsfile
\end
```

The main difference is the lack of LaTeX commands. The crucial difference is in the first two lines. There we simply `\input mfpic` and we turn on METAPOST support with the `\usemetapost` command instead of an option to `\usepackage`.

Since `\usepackage` and its options don't exist in plain TeX, all those features that we select with options in LaTeX must be selected by some command in plain. For example, the mplabels option is replaced with the command `\usemplabels` (which can also be used in LaTeX).

Also, plain TeX doesn't have environments, so instead of `\begin{mfpic}` we just use `\mfpic` and instead of `\end{mfpic}` we use `\endmfpic`.

The external processing is essentially the same:

```
pdftex plfirst
mpost myfigs
pdftex plfirst
```

should produce `plfirst.pdf` with the same picture of an ellipse.

11.2 MFPIC without PDF

If we wish to use nonPDF versions of LaTeX or plain TeX, the only difference is in the processing steps. To process `first.tex` with LaTeX, run the command

```
latex first
```

followed by

```
mpost myfigs
```

followed by latex again.

```
latex first
```

Then run the dvi processor of your choice. It should be one that can successfully handle eps figures (or at least the simple eps produced by METAPOST). Certainly DVIPS can do it:

```
dvips first
```

will produce `first.ps`. The file.ps file can be viewed with GSVIEW or printed, or converted to PDF with some distillation program like PS2PDF. Also DVIPDFM (if properly configured) can be used convert the `.dvi` file to PDF.

11.3 MFPIC without METAPOST

MFPIC can produce figures using METAFONT instead of METAPOST. What it does is work with METAFONT to produce a made-to-order font, where each picture is a large character in that font.

Since pdfTeX and pdfLaTeX do not work well with the fonts produced by METAFONT, and many PDF viewers don't display them well anyway, I do not recommend using MFPIC

to produce PDF without turning on METAPOST support. However, all dvi viewers and DVIPS *do* work well with such fonts, so it can make sense to use MFPIC with METAFONT *if* you don't need the features that METAPOST enables: color and rotation of labels. One advantage of doing this is the smaller number of files produced. If there are 100 MFPIC figures in a document, METAPOST produces 100 files (apart from a couple of temporary files and the `.log` file), but the METAFONT procedure produces only four files no matter how many figures are present.

To use MFPIC without METAPOST, omit the metapost option or the `\usemetapost` command. If you want a visible reminder of the fact that METAFONT is being used, you can use the metafont option or the `\usemetafont` command. Of course, you may not use mplabels without METAPOST. You may use the color commands and options, but the only colors actually produced will be black and white (and occasionally a pattern of pixels that simulate gray). The processing steps are different. After

```
latex first   (or tex plfirst)
```

run METAFONT:

```
mf myfigs
```

This should produce three files: `myfigs.log`, `myfigs.tfm`, and `myfigs.600gf`. The last one (which might have a different number on your system) is called a *generic font* (GF) file and contains the bitmap descriptions. If the file produced is `myfigs.2602gf`, and the `.tfm` is not produced, that indicates a configuration problem with your system that we'll get to later. If this did work, one needs to convert the GF file to a PK font file, the standard format for bitmap fonts in the TEX world. This may be done with

```
gftopk myfigs.600gf
```

Some systems may require you to name the output file on the command line:

```
gftopk myfigs.600gf myfigs.600pk
```

And some systems may require the extension to be simply `.pk`:

```
gftopk myfigs.600gf myfigs.pk
```

Finally, some systems may have a MAKEPK or MKTEXPK command that can be used in place of the combination of METAFONT and GFTOPK. You'll have to check what your system has and what its usage might be, and what it might do with the PK file produced.

After the above, one again runs '`latex first`' (or '`tex plfirst`'), and then the `.dvi` can be viewed or processed with dvips. The two files `myfigs.log` and `myfigs.600gf` can be deleted; only `myfigs.tfm` and `myfigs.600pk` are needed. If the viewed image shows the pictures at a far different size than you expect, this can also indicate a configuration problem.

Some systems permit on-the-fly creation of PK files by various `.dvi` processing programs. It is not wise to allow this to happen when working with MFPIC. The problem is that this automatic creation process is *not* repeated when a figure is edited unless the old PK files are deleted, and it may take some hunting to even locate them. One should *always* follow the METAFONT step with the GFTOPK step. You might even want to write a batch script or makefile to ensure that this happens.

Another problem (more an annoyance) that can occur comes from the behavior of most dvi viewers: most will reload a `.dvi` file if they detect that it has changed (or if asked to), but none that I know of will reload any fonts even if they have changed. So if one is going through

a edit-compile-view cycle involving MFPIC figures, one usually has to close the viewer and open it again before one can see changes that were made in the figures after starting the viewer. It is also possible that PK fonts are cached and shared by other programs, so you may need to close other programs to ensure the cache is cleared and the new figures loaded.

11.4 METAFONT configuration problems

To diagnose these problems it important to know something about *printer modes*. METAFONT produces bitmap images of characters. This means a description of a block of pixels, telling which ones are black and which are white. If the description says that 60 pixels in a row are black, that produces a thin black line. How long that line is depends on the size of a printer's pixels. For the LaserJet IV, there are 600 pixels to the inch, so 60 pixels makes 1/10 of an inch. The LaserJet II, however, has 300 pixels to the inch, so 60 pixels is 1/5 of an inch long. What METAFONT needs in order to produce an image that is the correct size is (at a minimum) the *resolution* of the intended printer. This is typically reported in DPI (dots per inch) and METAFONT keeps the value in the variable `pixels_per_inch`.

As part of the configuration of your DVI viewer or of DVIPS you may have needed to select a printer from a list, or edit a line in some configuration file (e.g., `config.ps`). What was going on then was the assigning of a default METAFONT printer mode. There is a file on most TEX systems named `modes.mf` which assigns symbolic names to a set of parameters that enable METAFONT to tune its output to a particular printer. For example, the LaserJet IV is given the name '`ljfour`' and that name is associated with the value 600 for `pixels_per_inch`. In order to tell METAFONT to make output for the LaserJet IV, one can put that information on the command line:

```
mf \mode:=ljfour; input myfigs
```

Your operating system or TEX distribution may require you to quote the backslash in the above command.

There is a system for making the selection of the correct mode semi-automatic, not requiring a command line specification. Near the end of `modes.mf` is a line similar to

```
localfont:=ljfour;
```

This is intended to equate the symbolic name `localfont` with the user's default printer. If the LaserJet IV is your default printer, the line above would be the correct one. If it is not, then that line should be changed. This can be done with an ordinary text editor, or your TEX system may have a configuration utility to take care of it.

If you say "`mf myfigs`" on an MFPIC file `myfigs.mf`, MFPIC's internal code will detect that no mode was defined on the command line. It will then check if `localfont` is defined and if so, use that for the printer mode. If that fails, it will try to select `ljfour`. If even that is unknown, MFPIC will define its own generic 600 DPI mode.

MFPIC doesn't need to know all the parameters associated to a printer mode, only the value of `pixels_per_inch`. If you get a GF file that indicates an incorrect DPI value for your printer, you should arrange for the line in `modes.mf` that sets `localfont` to be corrected. At the very least it should equate `localfont` to a name defined in `modes.mf` and associated to a printer with the same DPI as yours. After changing `modes.mf`, you need to run whatever programs your TEX system requires to remake the METAFONT format.