Pictures in TEX with Metafont and MetaPost

by
Dr Thomas E. Leathrum
Geoffrey Tobin*
Daniel H. Luecking†

2002/12/18

I. Introduction

1. Why?

Tom got the idea for MFPIC‡ mostly out of a feeling of frustration. Different output mechanisms for printing or viewing TEX DVI files each have their own ways to include pictures. More often than not, there are provisions for including PostScript data into a DVI file using TEX `\special`'s. However, this technique seems far from TEX's ideal of device-independence, and besides, different TEX output drivers handle these `\special`'s in different ways. The same problems arise with including TPIC `\special`'s.

LATEX's `picture` environment has a hopelessly limited supply of available objects to draw—if you want to draw a graph of a polynomial curve, you're out of luck.

There is, of course, PICTEX, which is wonderfully flexible and general, but its most obvious feature is its speed—or rather lack of it. Processing a single picture in PICTEX can often take several seconds.

It occurred to Tom that it might be possible to take advantage of the fact that METAFONT is *designed* for drawing things. The result of pursuing this idea is MFPIC, a set of macros for TEX and METAFONT which incorporate METAFONT-drawn pictures into a TEX file.

With the creation of METAPOST by John Hobby, and availability of free PostScript interpreters like Ghostscript, some MFPIC users wanted to run their MFPIC output through METAPOST, to produce PostScript pictures. Moreover, users wanted to be able to use pdfTEX, which does not get along well with PK fonts, but is quite happy with METAPOST pictures. Unfortunately `grafbase.mf`, which contained the METAFONT macros responsible for processing MFPIC's output, was far too pixel-oriented for METAPOST. A new file, `grafbase.mp` was created, based very heavily on `grafbase.mf` but compatible with METAPOST. Now when an MFPIC output file says `input grafbase`, either METAFONT or METAPOST may be run on it, and each program will select its own macros, and produce (nearly) the same picture.

With the extra capabilities of PostScript (e.g., color) and the corresponding abilities of METAPOST, there was a demand for some MFPIC interface to access them. Consequently, switches (options) have been added to access some of them. When these are used, output files may no longer be compatible with METAFONT.

MFPIC **Version: 0.6a beta.**

* `G.Tobin@latrobe.edu.au`

† `luecking@uark.edu`

‡ If you're wondering how to pronounce 'MFPIC': I always say 'em-eff-**pick**', spelling the first two letters. This explains expressions like '**an** MFPIC file'. —DHL.

2. Author.

MFPIC was written primarily by Tom Leathrum during the late (northern hemisphere) spring and summer of 1992, while at Dartmouth College. Different versions were being written and tested for nearly two years after that, during which time Tom finished his Ph.D. and took a job at Berry College, in Rome, GA. Between fall of 1992 and fall of 1993, much of the development was carried out by others. Those who helped most in this process are credited in the Acknowledgements.

The addition of METAPOST support was carried out by Dan Luecking around 1997–99.

3. Manifest.

Thirty-three files are included in this MFPIC distribution. See `manifest.txt` for a list and a brief explanation of each. Only five are actually needed for full access to MFPIC's capabilities: `mfpic.tex`, `mfpic.sty` (the latter needed only for LaTeX's `\usepackage`), `grafbase.mf` (needed only if METAFONT will be processing the figures), `grafbase.mp` and `dvipsnam.mp` (needed only if METAPOST will be the processor).

II. Setting Up and Processing.

Setting up TeX and METAFONT to process these files will, to an extent, depend on your local installation. The biggest problem you are likely to have, regardless of your installation, will be convincing TeX and its output drivers to find METAFONT's output files. You should do whatever is necessary to insure that TeX looks in the current directory for `.tfm` files, and that your dvi driver/viewer looks in the current directory for `.pk` files.

1. The Process

Here is an example of the process: for the sample file `pictures.tex`, first run TeX on it (or run LaTeX on `lapictures.tex`). You may see a message from MFPIC that there is no file `pics.tfm`, but TeX will continue processing the file anyway. When TeX is finished, you will now have a file called `pics.mf`. This is the METAFONT file containing the descriptions of the pictures for `pictures.tex`. You need to run METAFONT on `pics.mf`, with `\mode:=localfont` set up. (Read your METAFONT manual to see how to do this.*) This produces a `pics.tfm` file and a GF file with a name something like `pics.600gf`. The actual number may be different and the extension may get truncated on some file systems. Then you run GFTOPK on the GF file to produce a PK font file. (Read your GFTOPK manual on how to do this.) Now that you have the font and font metric files generated by METAFONT, reprocess the file `pictures.tex` with TeX. The resulting DVI file should now be complete, and you should be able to print and view it at your computer (assuming your viewer and print driver have been set up to be able to find the PK font generated from `pics.mf`).

It is not advisable to rely on automatic font generation to create the `.tfm` and `.pk` files. (Different systems do this in different ways, so here I will try to give a generic explanation.)

* If you are new to running METAFONT, the document *Metafont for Beginners*, by Geoffrey Tobin, is a good start. Fetch `CTAN/info/metafont-for-beginners.tex`. "CTAN" means the Comprehensive TeX Archive Network. You can find the mirror nearest you by pointing your browser at `http://www.ctan.org/`.

The reason: later editing of a figure will require new files to be built, and most automatic systems will *not* remake the files once they have been created. This is not so much a problem with the `.tfm`, as MFPIC never tries to load the font if the `.tfm` is absent and therefore no `.tfm`-making should ever be triggered. However, if you forget to run GFTOPK, then try to view your resulting file, you may have to search your system and delete some automatically generated `.pk` file (they can turn up in *very* strange places) before you can see any later changes. I suggest you write a shell script (batch file) that (1) runs METAFONT, (2) runs GFTOPK if step 1 returns no error, (3) deletes the `.tfm` if it exists, but the `.pk` file does not. That way, if anything goes wrong, the `.dvi` will not contain the font (MFPIC will draw a rectangle and the figure number in place of the figure).

These processing steps—processing with TEX, processing with METAFONT/GFTOPK, and reprocessing with TEX—may not always be necessary. In particular, if you change the TEX document without making any changes at all to the pictures, then there will be no need to repeat the METAFONT steps.

If you use MFPIC with the metapost option, then replace the METAFONT/GFTOPK steps with the single step of running METAPOST. (Read your METAPOST documentation on how to do this.*)

There are also somewhat subtle circumstance under which you can skip the second TEX step—if you change the picture in such a way as *not* to affect the font metric file (METAFONT case) or bounding box (METAPOST case), then you do not have to reprocess with TEX, because the original metric used previously will put the pictures in the right places. However, the post processing (DVIPS, for example) would still have to be repeated. If pdfTEXis used, that would also have to be repeated, since it is effectively also a post-processor. The exact cirumstances where metrics and bounding box are unchanged is rather involved, so it is recommended that you always repeat the TEX step if changes are made to a figure. However, there is one certain case: if you do not use the option mplabels, and if only text labels are added to a picture, then only one pass through TEX is required, and no additional runs of METAFONT or METAPOST.

2. How It Works.

When you run TEX on the file `pictures.tex`, TEX generates a file `pics.mf` (or `pics.mp`). This file is formed by `\write` commands in the `mfpic` macros. The user should never have to read or change the file `pics.mf` directly—the MFPIC macros take care of it.

The enterprising user can determine by examining the MFPIC drawing macros, that they translate almost directly into similar METAFONT/METAPOST commands, defined in `grafbase.mf`/`.mp`. The `\tlabel`'s and `\tcaption`'s, however, are placed on the graph by TEX, not METAFONT (except when options metapost and mplabels are both in effect, in which case METAPOST arranges the labels).

* The document *Some experiences on running Metafont and MetaPost*, by Peter Wilson, can be useful for beginners. Fetch `CTAN/info/metafp.pdf`.

III. Options.

There are now several options to the MFPIC package. These can be listed in the standard LaTeX `\usepackage` optional argument, or can be turned on with certain provided commands (the only possibility for plain TeX). Some options can be switched off and on throughout the document. Here we merely list them and provide a general description of their purpose. More details may be found later in the discussion of the features affected. The headings below give the option name, the alternative macro and, if available, the command for turning off the option.

1. metapost, `\usemetapost`

Selects METAPOST as the figure processor and makes specific features available. It changes the extension used on the output file to `.mp` to signal that it can no longer be processed with METAFONT. There is also a metafont option (command `\usemetafont`), but it is redundant, as METAFONT is the default. Either command must come before the `\opengraphsfile` command (see section FILES AND ENVIRONMENTS). They should not be used together in the same document. (Actually, this hasn't ever been tested, but it definitely wasn't taken into consideration in writing the macros.) If the command form `\usemetapost` is used in a LaTeX2$_\varepsilon$ document, it must come in the preamble. Because of the timing of actions by the GRAPHICS package's `pdftex.def` and by BABEL,* when pdfLaTeX is used MFPIC should be loaded and `\usemetapost` (if used) declared before BABEL is loaded.

2. mplabels, `\usemplabels`, `\nomplabels`

Causes all `\tlabel` commands to write their contents to the output file. It has no effect on `\tcaption` commands. In this case labels are handled by METAPOST, and can be rotated. It requires METAPOST, and must come after METAPOST has been selected or it will produce an error and be ignored (METAFONT cannot handle labels). Otherwise the commands can come anywhere and affect subsequent `\tlabel` commands, but it is recommended that you *not* mix the two methods of handling labels in a single `mfpic` environment, and that you *only* use mplabels and truebbox (see below) together. When this is in effect, the labels become part of the figure: they may be clipped if the clip option is in effect, and they contribute to the bounding box if truebbox is in effect.

The user is responsible for adding the appropriate `verbatimtex` header to the output file if necessary. For this purpose, there is the `\mfpverbtex` command, see the section LABELS AND CAPTIONS. If the label text contains only valid plain TeX macros, there is generally no need for a `verbatimtex` preamble at all. If you add a `verbatimtex` preamble of LaTeX code take care to make sure METAPOST calls LaTeX (for example, by setting the environmental variable `TEX` to `latex` in the command shell of your operating system.).

* At least as of this writing.

3. truebbox, \usetruebbox, \notruebbox

Normally METAPOST produces an output file with the exact bounding box of the figure. By default, MFPIC overrides this and sets the bounding box to the dimensions specified by the \mfpic command. This is be needed if TEX is to handle \tlabel commands correctly. If no \tlabel commands are present, or if mplabels is used, then it is reasonable to let METAPOST have its way. That is what this option does. The commands can come anywhere *outside* any mfpic environment (because they change certain aspects of the opening code of the command \mfpic), and affect all subsequent figures. This option has no effect with METAFONT, but should cause no errors.

Again, it is recommended to use either both truebbox and mplabels or neither of them. Actually, either one *should* work without the other but the code depends on internals of the figure placement packages used, which could possibly change. Such a change could, in fact, break the current code even if you follow this recommendation. If you should get a message similar to "Undefined control sequence \Gin@llx", a possible fix is to redefine the command \getmfpicoffset (see section FOR POWER USERS ONLY) to be a no-op (\def\getmfpicoffset{}).

4. clip, \clipmfpic, \noclipmfpic

Causes all parts of the figure outside the rectangle specified by the \mfpic command to be removed. The commands can come anywhere. If issued inside an mfpic environment they affect the current figure only. Otherwise all subsequent figures are affected. Note: this is a rather rudimentary option. If mplabels is in effect, and the labels extend beyond the nominal bounds of the figure, they may be clipped off. It has an often unexpected interaction with truebbox. When both are in effect, METAPOST will produce a "true" bounding box that is the intersection of two bounding boxes: the true one *without clipping*, and the box specified in the \mfpic command. It is possible that the actual figure will be much smaller (even empty!). This is a property of the METAPOST clip command and we know of no way to avoid it.

5. centeredcaptions, \usecenteredcaptions, \nocenteredcaptions

Causes multiline captions created by \tcaption to have all lines centered. This has no effect on the normal LATEX \caption command.* The commands can be issued anywhere. If inside an mfpic environment they should come before the \tcaption and affect only it, otherwise they affect all subsequent figures.

6. debug, \mfpicdebugtrue, \mfpicdebugfalse

Causes MFPIC to write a rather large amount of information to the .log file and sometimes to the terminal. Debug information generated by mfpic.tex *while loading* is probably of interest only to developers, but can be turned on by giving a definition to the command \mfpicdebug prior to loading.

* This writer [DHL] feels that \tcaption is too limited and users ought to apply the caption by other means, such as LATEX's \caption command, outside the mfpic environment.

7. draft, final, nowrite, \mfpicdraft, \mfpicfinal, \mfpicnowrite

Under the metapost option, the various macros that include the EPS files emit rather large amounts of confusing error messages when the files don't exist (especially in LaTeX). For this reason, before each picture is placed, MFPIC checks for the existence of the graphic before trying to include it. However, on some systems checking for existence can be very slow, because the entire TeX search path may need to be checked. Therefore, MFPIC doesn't even attempt any inclusion on the first run. The first run is detected by the non-existence of ⟨*file*⟩`.1`, where ⟨*file*⟩ is the name given in the `\opengraphsfile` command (see section FILES AND ENVIRONMENTS). These options can be used to override this automatic detection. The command versions must come *before* the `\opengraphsfile` command.

These options might be used if, for example, the first figure has an error and is not created by METAPOST, but you would like MFPIC to go ahead and include the remaining figures. Then use final. It can also be used to overrule a LaTeX global draft option. Or if ⟨*file*⟩`.1` exists, but other figures still have errors and you would like several runs to be treated as first runs until METAPOST has stopped issuing error messages. Then use draft. These commands also work under the metafont option, but time and error messages are less of an issue then. If all the figures have been created and debugged, some time might be saved (with either metafont or metapost) by not writing the output file again, then nowrite can be used.

8. OPTION SCOPING RULES

Some of these options merely change TeX behavior, others write information to the output file for METAFONT or METAPOST. Changes in TeX behavior obey the normal TeX grouping rules, the information written to the output file obeys METAFONT grouping rules. Since each `mfpic` environment is both a TeX group and (corresponds to) a METAFONT group, the following always holds: use of one of the command forms inside of an `mfpic` environment makes the change local to that environment.

For options that affect only TeX, the command forms have effects local to *any* TeX groups. For options that send information to METAFONT, the commands' effects are *only* restricted by `\mfpic` environments. The commands influencing only TeX are the following: `\usemplabels`, `\nomplabels`,† `\usecenteredcaptions`, `\nocenteredcaptions`, `\mfpicdebugtrue`, and `\mfpigdebugfalse`. The commands affecting only the output file are `\clipmfpic` and `\noclipmfpic`.

The following commands are special: `\usemetapost`, `\usemetafont`, `\usetruebbox`, `\notruebbox`, `\mfpicdraft`, `\mfpicfinal`, and `\mfpicnowrite`. Their effects are always global, partly because they should not be issued inside any `mfpic` environment (the start-up code of `\mfpic` needs to know their settings).

† The macros `\usemplabels` and `\nomplabels` obviously affect the output file, but they do not write anything to the output file that changes any definition or parameter value, and so no scoping issues are involved.

IV. The Macros.

1. Files and Environments.

`\opengraphsfile{`⟨*file*⟩`}`
  ...
`\closegraphsfile`

These macros open and close the METAFONT or METAPOST file which will contain the pictures to be included in this document. The name of the file will be ⟨*file*⟩`.mf` (or ⟨*file*⟩`.mp`). If the ⟨*file*⟩ parameter is changed, you will have to reprocess the TEX file after processing the output file. Do *not* specify the extension, which is added automatically.

Note: This command will cause ⟨*file*⟩`.mf` or ⟨*file*⟩`.mp` to be overwritten if it already exists, so be sure to consider that when selecting the name. Repeating the running of TEX will overwrite the file created on previous runs, but that should be harmless. For if no changes are made to `mfpic` environments, the identical file will be recreated, and if changes have been made, then you want the file to be replaced with the new version.

`\mfpic[`⟨*xscale*⟩`][`⟨*yscale*⟩`]{`⟨*xmin*⟩`}{`⟨*xmax*⟩`}{`⟨*ymin*⟩`}{`⟨*ymax*⟩`}`
  ...
`\endmfpic`

These macros open and close the `mfpic` environment in which the rest of the macros below make sense. The `\mfpic` macro also sets up the local coordinate system for the picture. The ⟨*xscale*⟩ and ⟨*yscale*⟩ parameters establish the length of a coordinate system unit, as a multiple of the TEX dimension `\mfpicunit`. If neither is specified, both are taken to be 1 (i.e., each coordinate system unit is 1 `\mfpicunit`). If only one is specified, then they are assumed to be equal. The ⟨*xmin*⟩ and ⟨*xmax*⟩ parameters establish the lower and upper bounds for the x-axis coordinates; similarly, ⟨*ymin*⟩ and ⟨*ymax*⟩ establish the bounds for the y-axis. These bounds are expressed in local units—in other words, the actual width of the picture will be $(\langle xmax\rangle - \langle xmin\rangle) \cdot \langle xscale\rangle$ times `\mfpicunit`, its height $(\langle ymax\rangle - \langle ymin\rangle) \cdot \langle yscale\rangle$ times `\mfpicunit`, and its depth zero. One can scale all pictures uniformly by changing `\mfpicunit`, and scale an individual picture by changing ⟨*xscale*⟩ and ⟨*yscale*⟩. After loading MFPIC, `\mfpicunit` has the value `1pt`.

Note: Changing `\mfpicunit` or the optional parameters will scale the coordinate system, but not the values of certain parameters that are defined in absolute units. Examples of these are the default width of the drawing pen, the default lengths of arrowheads, the default sizes of dashes and dots, etc. If you wish, you can set these to multiples of `\mfpicunit`, but it is difficult (and probably unwise) to get them to scale along with the scale parameters.

In addition to establishing the coordinate system, these scales and bounds are used to establish the metric for the METAFONT character or bounding box for the METAPOST figure described within the environment. If any of these parameters are changed, the `.tfm` file (METAFONT) or the bounding box (METAPOST) will be affected, so you will have to reprocess the TEX file after processing the `.mf` or `.mp` file.

`\mfpicnumber{`⟨*num*⟩`}`

Normally, `\mfpic` assigns the number 1 to the first `mfpic` environment, after which the number is increased by one for each new `mfpic` environment. This number is used internally to include the picture. It is also transmitted to the output file where it is used as the argument to a `beginmfpic` command. In METAFONT this number becomes the position of the character in the font file, while in METAPOST it is the extension on the graphic file that is output. The above command tells MFPIC to ignore this sequence and number the next `mfpic` figure with ⟨*num*⟩ (and the one after that ⟨*num*⟩ + 1, etc.). It is up to the user to make sure no number is repeated, as no checking is done. Numbers greater than 255 may cause errors, as TEX assumes that characters are represented by 8-bit numbers. If the first figure is to be numbered something other than 1, then, under the metapost option, this command should come before `\opengraphsfile`, as that command checks for the existence of the first numbered figure to determine if there are figures to be included.

`\begin{mfpic}...\end{mfpic}`

In LATEX, instead of using the `\mfpic` and `\endmfpic` macros, you may prefer to use `\begin{mfpic}` and `\end{mfpic}`. This is by no means required: LATEX has been designed so that `\begin{command}` effectively means `\command`, and `\end{command}` effectively means `\endcommand`, for any TEX command.

The sample file `lapictures.tex` provided with MFPIC illustrates this use of an `mfpic` environment in LATEX.

Note that the `\opengraphsfile` and `\closegraphsfile` macros should be used under those names in LATEX too, as they do not possess a `\command...\endcommand` structure.

The rest of the MFPIC macros do not affect the font metric file (⟨*file*⟩`.tfm`), and so if these commands are changed or added in your document, you will not have to repeat the third step of processing (reprocessing with TEX) to complete your TEX document. The same is true when option metapost is selected, except under pdfTEX or pdfLATEX, and except when the truebbox option is used.

For the remainder of the macros, the numerical parameters are expressed in the units of the local coordinate system specified by `\mfpic`, unless otherwise indicated.

2. Figures.

2.1 METAFONT *Pairs.*

Since many of the arguments of the MFPIC drawing commands are sent to METAFONT to be interpreted, it's useful to know something about METAFONT concepts.

In particular, METAFONT has `pair` objects, which may be constants or variables. Pair constants have the form `(`x`,`y`)`. Pairs are two-dimensional rectangular (cartesian) quantities, and are clearly useful for representing both points and vectors on the plane.

Moreover, we herein often represent each pair by a brief name, such as p, v or c, the meanings of which are usually obvious in the context of the macro. The succinctness of this notation also helps us to think geometrically rather than only of coordinates.

METAPOST has these same concepts, but also has color objects, which may also be constants or variables. Color constants have the form `(`r`,`g`,`b`)` where r, g, and b are numbers

between 0 and 1 determining the relative proportions of red, green and blue in the color (rgb model). A color variable is a name, like `magenta` or `RoyalBlue` (predefined), or a color function like `cmyk(`x`,`y`,`z`,`w`)` which is defined to convert cmyk values into METAPOST's native rgb model.

Some commands depend on the value of separately defined parameters. All these parameters are initialized when MFPIC is loaded. In the following descriptions we give the initial value of all the relevant parameters. To save having to repeat this for each command: when METAPOST output is selected, all the commands that draw curves draw them in `drawcolor`; those that fill shapes fill them in `fillcolor`; those that create hatch lines draw them in `hatchcolor`; those that create arrowheads draw them in `headcolor`; commands that use METAPOST to place text color it in `tlabelcolor`, and the `\gclear` command fills a contour with the special color `background`. All these colors are predefined and initialized to `black`, except `background` is initially `white`, but all can be changed by the user (see the COLORS section below).

2.2 *Points, Lines, and Rectangles.*

`\pointdef{`⟨*name*⟩`}(`x`,`y`)`

Defines a symbolic name for points and their coordinates. ⟨*name*⟩ is any legal TEX command name *without* the backslash; x and y are any numbers. For example, after the command `\pointdef{A}(1,3)`, `\A` expands to `(1,3)`, while `\Ax` and `\Ay` expand to `1` and `3`, respectively. Because of the way `\tlabel` is defined (see section LABELS AND CAPTIONS below), one cannot use `\A` to specify where to place a label (unless mplabels is in effect), but must use `(\Ax,\Ay)`. In most other commands, one can use `\A` where a pair or point is required.

`\point[`⟨*ptsize*⟩`]{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws small disks centered at the points ⟨p_0⟩, ⟨p_1⟩, and so on. If the optional argument ⟨*ptsize*⟩ is present, it determines the diameter of the disks, which otherwise equals the TEX dimension `\pointsize`, initially `2pt`. The disks have a filled interior if the command `\pointfilltrue` has been issued (the initial value), `\pointfillfalse` causes the interior to be erased and an outline drawn. The color of the filled circles is the predefined `fillcolor`, and the inside of the open circles is the predefined `background`.

`\plotsymbol[`⟨*size*⟩`]{`⟨*symbol*⟩`}{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws small symbols centered at the points ⟨p_0⟩, ⟨p_1⟩, and so on. The symbols must be given by name, and the available symbols are `Asterisk`, `Circle`, `Diamond`, `Square`, `Triangle`, `Star`, `SolidCircle`, `SolidDiamond`, `SolidSquare`, `SolidTriangle`, `SolidStar`, `Cross` and `Plus`. The names should be self-explanatory. The "Solid..." symbols are drawn in `fillcolor`, the open ones in `drawcolor`. The ⟨*size*⟩ defaults to `\pointsize` as in `\point` above. The difference between `Star` and `SolidStar` is not very evident unless different colors are used or the ⟨*size*⟩ is rather large (around `3pt` at printer resolutions, more for screen viewing). `Asterisk` consists of five line segments while `Star` is the standard closed, ten-sided polygon.

The difference between `\pointfillfalse\point`... and `\plotsymbol{Circle}`... is that the center of the circle will not be erased in the second version.

`\polyline{`⟨p_0⟩`,`⟨p_1⟩`,...}`
`\lines{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws the line segment with endpoints at ⟨p_0⟩ and ⟨p_1⟩, then the line segment with endpoints at ⟨p_1⟩ and ⟨p_2⟩, etc. The result is an open polygonal path through the specified points, in the specified order. `\polyline` and `\lines` mean the same thing.

`\polygon{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws a closed polygon with vertices at the specified points.

`\rect{`⟨p_0⟩`,`⟨p_1⟩`}`

Draws the rectangle specified by the points ⟨p_0⟩ and ⟨p_1⟩, these being any two opposite corners of the rectangle.

2.3 *Axes, Axis Marks, and Grids.*

[*As of version 0.5, axis handling was revamped. The three commands below were retained for backward compatibility, but alternatives were added with somewhat different behavior.*]

`\axes[`⟨*hlen*⟩`]`
`\xaxis[`⟨*hlen*⟩`]`
`\yaxis[`⟨*hlen*⟩`]`

Draw x- and y-axes for the coordinate system. The command `\axes` is equivalent to `\xaxis` followed by `\yaxis` which produce the obvious. The x- and y-axes extend the full width and height of the `mfpic` environment. The optional ⟨*hlen*⟩ sets the length of the arrowhead on each axis. The default is the value of the TEX dimension `\axisheadlen`, initially `5pt`. The shape of the arrowhead is determined as in the `\arrow` macro below. The color of the head is `headcolor`.

Unlike other commands that produce lines or curves, these do not respond to the prefix macros of the section Shape-Modifier Macros. They always draw a solid line (with an arrowhead unless `\axisheadlen` is `0pt`). They *do* respond to changes in the pen thickness (see `\penwd` in section Parameters).

`\axis[`⟨*hlen*⟩`]{`⟨*one-axis*⟩`}`
`\doaxes[`⟨*hlen*⟩`]{`⟨*axis-list*⟩`}`

These produce any of 6 different axes. The parameter ⟨*one-axis*⟩ can be `x` or `y`, to produce (almost) the equivalent of `\xaxis` and `\yaxis`; or it can be `l`, `b`, `r`, or `t` to produce an axis on the border of the picture (left, bottom, right or top, respectively). `\doaxes` takes a list of any or all of the six letters (with either spaces or nothing in between) and produces the appropriate axes. Example: `\doaxes{lbrt}`. The optional argument sets the length of the arrowhead. In the case of axes on the edges, the default is the value of `\sideheadlen`, which `mfpic.tex` initializes to `0pt`. For the x- and y-axis the default is `\axisheadlen` as in `\xaxis` and `\yaxis` above.

The commands `\axis{x}`, `\axis{y}`, and `\doaxes{xy}` differ from the old `\xaxis`, `\yaxis` and `\axes` in that these new versions respond to changes made by `\setrender` (see subsection *Changing the Default Rendering* of section SHAPE-MODIFIER MACROS). Moreover, prefix macros may be applied to `\axis` without error (see section SHAPE-MODIFIER MACROS): `\dotted\axis{x}` draws a dotted x-axis, but `\dotted\xaxis` produces an error (from METAFONT). (A prefix macro applied to `\doaxes` generates no error, but only the first axis in the list will be affected.)

The side axes are drawn by default with a pen stroke along the very edge of the picture (as determined by the parameters to `\mfpic`). This can be changed with the commands `\axismargin`, `\setaxismargins` and `\setallaxismargins`, described below.

Axes on the edges are drawn so that they don't cross each other. For example, `\doaxes{lbrt}` produces a perfect rectangle. If the x- and y-axis are drawn with `\axis` or `\doaxis`, then they will not cross the side axes. For this to work properly, all the following margin settings have to be done before the axes are drawn.

`\axismargin{`⟨*axis*⟩`}{`⟨*num*⟩`}`
`\setaxismargins{`⟨*num*⟩`}{`⟨*num*⟩`}{`⟨*num*⟩`}{`⟨*num*⟩`}`
`\setallaxismargins{`⟨*num*⟩`}`

The ⟨*axis*⟩ is one of the letters `l`, `b`, `r`, or `t`. `\axismargin` causes the given axis to be shifted *inward* by the ⟨*num*⟩ specified (in *graph* coordinates). The second command `\setaxismargins` takes 4 arguments, using them to set the margins starting with the left and proceeding anticlockwise. The last command sets all the axis margins to the same value.

A change to an axis margin affects not only the axis at that edge but also the three axes perpendicular to it. For example, if the margins are M_{lft}, M_{bot}, M_{rt} and M_{top}, then `\axis b` draws a line starting M_{lft} graph units from the left edge and ending M_{rt} units from the right edge. Of course, the entire line is M_{bot} units above the bottom edge. The margins are also respected by the x- and y-axis, but only when drawn with `\axis`. The old `\xaxis`, `\yaxis` and `\axes` ignore them.

Special effects can be achieved by lying to one axis about the other margins. For example, the left and right axis can be made to cross the bottom and top axis with

```
\setaxismargins{1}{0}{1}{0}
\doaxes{rl}
\setaxismargins{0}{1}{0}{1}
\doaxes{bt}
```

`\xmarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\tmarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\bmarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\ymarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\lmarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\rmarks[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`
`\axismarks{`⟨*axis*⟩`}[`⟨*len*⟩`]{`⟨*numberlist*⟩`}`

These macros place hash marks on the appropriate axes at the places indicated by

the values in the list. The optional ⟨*len*⟩ gives the length of the hash marks. If ⟨*len*⟩ is not specified, the TEX dimension `\hashlen`, initially `4pt`, is used. The marks on the x- and y-axes are centered on the respective axis; the marks on the border axes are drawn to the inside. Both these behaviors can be changed (see below). The commands may be repeated as often as desired. (The timing of drawing commands can make a difference as outlined in `mppicdoc.tex`.) The command `\axismarks{x}` is equivalent to `\xmarks` and so on for each of the six axes. (I would have used `\marks`, but eTEX makes that a primitive.)

The ⟨*numberlist*⟩ is normally a comma-separated list of numbers. In place of this, one can give a starting number, an increment and an ending number as in the following example:

```
\xmarks{-2 step 1 until 2}
```

is the equivalent of

```
\xmarks{-2,-1,0,1,2}
```

One must use exactly the words `step` and `until`. There must be spaces between, but the number of spaces is not significant.* Users should be aware that if any of the numbers are non-integral then due to natural round-off effects, the last value might be overshot and a mark not printed there.

`\setaxismarks{`⟨*axis*⟩`}{`⟨*pos*⟩`}`
`\setbordermarks{`⟨*lpos*⟩`}{`⟨*bpos*⟩`}{`⟨*rpos*⟩`}{`⟨*tpos*⟩`}`
`\setallbordermarks{`⟨*pos*⟩`}`
`\setxmarks{`⟨*pos*⟩`}`
`\setymarks{`⟨*pos*⟩`}`

These set the placement of the hash marks relative to the axis. The parameter ⟨*axis*⟩ is one of the letters `x`, `y`, `l`, `b`, `r`, or `t`, and ⟨*pos*⟩ must be one of the literal words `inside`, `outside`, `centered`, `onleft`, `onright`, `ontop` or `onbottom`. The second command takes four arguments and sets the position of the marks on each border. The third command sets the position on all four border axis to the same value. The last two commands are abbreviations for `\setaxismarks{x}{`⟨*pos*⟩`}` and `\setaxismarks{y}{`⟨*pos*⟩`}`, respectively.

Not all combinations make sense (for example, `\setaxismarks{r}{ontop}`). In these cases, no error message is produced: `ontop` and `onleft` are considered to be equivalent, as are `onbottom` and `onright`. The parameters `inside` and `outside` make no sense for the x- and y-axes, but if used `inside` means `ontop` for the x-axis and `onright` for the y-axis. These words are actually METAFONT numeric variables defined in the file `grafbase.mf`, and the variables `ontop` and `onleft`, for example, are given the same value.

* Experienced METAFONT programmers may recognize that anything can be used that is permitted in METAFONT's ⟨*forloop*⟩ syntax. Thus the given example can also be reworded `\xmarks{-2 upto 2}`, or even `\xmarks{2 downto -2}`

`\grid[`⟨*ptsize*⟩`]{`⟨*xsep*⟩`,`⟨*ysep*⟩`}`
`\gridpoints[`⟨*ptsize*⟩`]{`⟨*xsep*⟩`,`⟨*ysep*⟩`}`
`\lattice[`⟨*ptsize*⟩`]{`⟨*xsep*⟩`,`⟨*ysep*⟩`}`
`\gridlines{`⟨*xsep*⟩`,`⟨*ysep*⟩`}`
`\plrgrid{`⟨*rsep*⟩`,`⟨*anglesep*⟩`}`
`\plrpatch{`⟨*rmin*⟩`,`⟨*rmax*⟩`,`⟨*rsep*⟩`,`⟨*tmin*⟩`,`⟨*tmax*⟩`,`⟨*tsep*⟩`}`

`\grid` draws a dot at every point for which the first coordinate is an integer multiple of the ⟨*xsep*⟩ and the second coordinate is an integer multiple of ⟨*ysep*⟩. This behavior of `\grid` started with version 0.4.05. Before that, the points were drawn on the left border (and bottom border) and every ⟨*xsep*⟩ right of (⟨*ysep*⟩ above) those. The diameter of the dot is determined by ⟨*ptsize*⟩, the default is `.5pt`. The commands `\gridpoints and \lattice` are synonyms for `\grid`. `\gridlines` draws the horizontal and vertical lines through these same points.

`\plrgrid` fills the graph with circular arcs and radial lines. The arcs are centered at $(0, 0)$ and the lines emanate from $(0, 0)$ (even if $(0, 0)$ is not in the graph space). The corresponding METAFONT commands actually draw enough to cover the graph area and then clip them to the graph boundaries. If you don't want them clipped, use `\plrpatch` which draws arcs with radii starting at ⟨*rmin*⟩, stepping by ⟨*rsep*⟩ and ending with ⟨*rmax*⟩. Each arc goes from angle ⟨*tmin*⟩ to ⟨*tmax*⟩. It also draws radial lines with angles starting at ⟨*tmin*⟩, stepping by ⟨*tsep*⟩ and ending with ⟨*tmax*⟩. Each line goes from radius ⟨*rmin*⟩ to ⟨*rmax*⟩. If ⟨*rmax*⟩ − ⟨*rmin*⟩ doesn't happen to be a multiple of ⟨*rsep*⟩, the arc with radius ⟨*rmax*⟩ is drawn anyway. The same is true of the line at angle ⟨*tmax*⟩, so that the entire boundary is always drawn.

2.4 *Circles and Ellipses.*

`\circle{`⟨*c*⟩`,`⟨*r*⟩`}`

Draws a circle centered at the point ⟨*c*⟩ and with radius ⟨*r*⟩.

`\ellipse[`⟨*θ*⟩`]{`⟨*c*⟩`,`⟨r_x⟩`,`⟨r_y⟩`}`

Draws an ellipse with the x radius ⟨r_x⟩ and y radius ⟨r_y⟩, centered at the point ⟨*c*⟩. The optional parameter ⟨*θ*⟩ provides a way of rotating the ellipse by ⟨*θ*⟩ degrees counterclockwise around its center.

2.5 *Curves.*

`\curve[`⟨*tension*⟩`]{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws a METAFONT Bézier path through the specified points, in the specified order. The optional ⟨*tension*⟩ defaults to 1 and influences *how* smooth the curve is. The special value `infinity` (in fact, usually anything greater than about 10), makes the curve indistinguishable from `\polyline`. The higher the value of tension, the sharper the corners on the curve and the flatter the portions in between. METAFONT requires the tension to be larger than 0.75.

`\cyclic[`⟨*tension*⟩`]{`⟨p_0⟩`,`⟨p_1⟩`,...}`

Draws a cyclic (i.e., closed) METAFONT Bézier curve through the specified points, in the specified order. The ⟨*tension*⟩ is as in the `\curve` command.

Occasionally it is necessary to specify a sequence of points with *increasing* x coordinates and draw a curve through them. One would then like the resulting curve both to be smooth *and* to represent a function (that is, the curve always has increasing x coordinate, never turning leftward). This cannot be guaranteed with the `\curve` command unless the tension is `infinity`.

`\fcncurve[`⟨*tension*⟩`]{(`x_0`,`y_0`),(`x_1`,`y_1`),...}`

Draws a curve through the points specified. If the points are listed with increasing (or decreasing) x coordinates, the curve will also have increasing (resp., decreasing) x coordinates. The ⟨*tension*⟩ is a number equal to or greater than 1.0 which controls how tightly the curve is drawn. Generally, the larger it is, the closer the curve is to the polyline through the points. The default tension is 1.2. (For those who know something about METAFONT, this "tension" is not the same as the METAFONT notion of tension, nor the same as the tension in the `\curve` command, but it functions in a similar fashion.)

2.6 *Circular Arcs.*

`\arc[`⟨*format*⟩`]{...}`

Draws a circular arc specified as determined by the ⟨*format*⟩ optional parameter—this macro is unusual in that the optional ⟨*format*⟩ parameter determines the format of the other parameter, as indicated below. The user is responsible for ensuring that the parameter values make geometric sense.

`\arc[s]{`⟨p_0⟩`,`⟨p_1⟩`,`⟨*sweep*⟩`}`

(*Point-Sweep Form —this is the default format.*) Draws a circular arc starting from the point ⟨p_0⟩, ending at the point ⟨p_1⟩, and covering an arc angle of ⟨*sweep*⟩ degrees, measured counterclockwise around the center of the circle. If, for example, the points ⟨p_0⟩ and ⟨p_1⟩ lie on a horizontal line with ⟨p_0⟩ to the *left*, and ⟨*sweep*⟩ is between 0 and 360 (degrees), then the arc will sweep *below* the horizontal line (in order for the arc to be counterclockwise). A negative value of ⟨*sweep*⟩ gives a clockwise arc from ⟨p_0⟩ to ⟨p_1⟩.

`\arc[t]{`⟨p_0⟩`,`⟨p_1⟩`,`⟨p_2⟩`}`

(*Three-Point Form.*) Draws the circular arc which passes through all three points given, in the order given.

`\arc[p]{`⟨c⟩`,`⟨θ_1⟩`,`⟨θ_2⟩`,`⟨r⟩`}`

(*Polar Form.*) Draws the arc of a circle with center ⟨c⟩ starting at the angle ⟨θ_1⟩ and ending at the angle ⟨θ_2⟩, with radius ⟨r⟩. Both angles are measured counterclockwise from the positive x axis.

`\arc[c]{`⟨c⟩`,`⟨p_1⟩`,`⟨θ⟩`}`

(*Center-Point Form.*) Draws the circular arc with center ⟨c⟩, starting at the point ⟨p_1⟩, and sweeping an angle of ⟨θ⟩ around the center from that point. (This is actually MFPIC's internal way of handling arcs—all other formats are translated to this format before drawing.)

2.7 *Other Figures.*

`\turtle{`⟨p_0⟩`,`⟨v_1⟩`,`⟨v_2⟩`,...}`

Draws a line segment, starting from the point ⟨p_0⟩, and extending along the (2-dimensional vector) displacement ⟨v_1⟩. It then draws a line segment from the previous segment's endpoint, along displacement ⟨v_2⟩. This continues for all listed displacements, a process similar to "turtle graphics".

`\sector{`⟨c⟩`,`⟨r⟩`,`⟨θ_1⟩`,`⟨θ_2⟩`}`

Draws the sector, from the angle ⟨θ_1⟩ to the angle ⟨θ_2⟩ inside the circle with center at the point ⟨c⟩ and radius ⟨r⟩, where both angles are measured in degrees counterclockwise from the direction parallel to the x axis. The sector forms a closed path. *Note*: `\sector` and `\arc[p]` have the same parameters, but *in a different order*.*

2.8 *Bar Charts and Pie Charts.*

`\barchart[`⟨*start*⟩`,`⟨*sep*⟩`,`⟨r⟩`]{`⟨*h-or-v*⟩`}{`⟨*list*⟩`}`
`\chartbar{`⟨*num*⟩`}`

The macro `\barchart` does not actually draw anything, but computes the `\chartbar` rectangles. ⟨*h-or-v*⟩ should be `v` if you want the bars to extend vertically from the x-axis, or `h` if they should extend horizontally from the y-axis. ⟨*list*⟩ should be a comma-separated list of numbers giving the coordinates of the end of each bar (the end that is not on the axis). The rest of this description refers to the `v` case; the `h` case is analogous.

By default the bars are 1 graph unit wide, and the base of the nth bar is the interval $[n-1, n]$. The optional parameter consists of three numeric parameters separated by commas. ⟨*start*⟩ is the x-coordinate of the left edge of the first bar, ⟨*sep*⟩ is the distance between the left edges of adjacent bars, and ⟨r⟩ is the fraction of ⟨*sep*⟩ occupied by each bar. The default behavior corresponds to `[0,1,1]`.

The fraction ⟨r⟩ should be between -1 and 1, a negative value indicating that the left edge referred to above is actually the right edge. For example, if one bar chart is created with `\barchart[1,-.4]{v}{..}` and another with `\barchart[1,1,.4]{v}{..}` having the same number of bars, then first will have its nth bar from $n-.4$ to n, while the second will have its nth bar adjacent to that one, from n to $n+.4$. This makes it easy to draw bars side-by-side for comparison.

After a `\barchart` command has been issued, the individual bars may be drawn with `\chartbar{1}`, `\chartbar{2}`, etc. These are closed rectangles and can be filled, shaded,

* This apparently was a mistake which we now have to live with so as not to break existing `.tex` files.

hatched, etc., using appropriate prefix macros (see section SHAPE-MODIFIER MACROS, below).

`\piechart[`⟨*dir*⟩⟨*angle*⟩`]{`⟨*c*⟩`,`⟨*r*⟩`}{`⟨*list*⟩`}`
`\piewedge[`⟨*spec*⟩⟨*trans*⟩`]{`⟨*num*⟩`}`

The macro `\piechart` also does not draw anything, but computes the `\piewedge` regions described below. The first part of the optional parameter, ⟨*dir*⟩, is a single letter which may be either `c` or `a` which stand for *clockwise* or *anticlockwise*, respectively. It is common to draw piecharts with the largest wedge starting at 12 o'clock (angle 90 degrees) and successive wedges clockwise from there. This is the default. You can change the starting angle from 90 with the ⟨*angle*⟩ parameter, and the change the direction to counter-clockwise by specifying `a` for ⟨*dir*⟩. It is also traditional to arrange the wedges from largest to smallest, except there is often a miscellaneous category which is usually last and may be larger than some others. Therefore `\piechart` makes no attempt to sort the data. The data is entered as a comma separated ⟨*list*⟩ of positive numbers in the second required parameter. These are only used to determine the relative sizes of the wedges and are not printed anywhere. The first required parameter should contain a pair ⟨*c*⟩ for the center and a positive number ⟨*r*⟩ for the radius, separated by a comma.

After a `\piechart` command has been issued, the individual wedges may be drawn, filled, etc., using `\piewedge{1}`, `\piewedge{2}`, etc. Without the optional argument, the wedges are located according to the arguments of the last `\piechart` command. The optional argument to `\piewedge` can override this. The parameter ⟨*spec*⟩ is a single letter, which can be `x`, `s` or `m`. The `x` stands for *exploded* and it means the wedge is moved directly out from the center of the pie a distance ⟨*trans*⟩. ⟨*trans*⟩ should then be a pure number and is interpreted as a distance in graph units. The `s` stands for *shifted* and in this case ⟨*trans*⟩ should be a pair of the form `(`⟨*dx*⟩`,`⟨*dy*⟩`)` indicating the wedge should be shifted ⟨*dx*⟩ horizontally and ⟨*dy*⟩ vertically (in graph units). The `m` stands for *move to*, and ⟨*trans*⟩ is then the absolute coordinates `(`⟨*x*⟩`,`⟨*y*⟩`)` in the graph where the point of the wedge should be placed.

2.9 *Polar Coordinates to Rectangular.*

`\plr{(`⟨r_0⟩`,`⟨θ_0⟩`), (`⟨r_1⟩`,`⟨θ_1⟩`), ...}`

Replaces the specified list of polar coordinate pairs by the equivalent list of rectangular (cartesian) coordinate pairs. Through `\plr`, commands designed for rectangular coordinates can be applied to data represented in polar coordinates—and to data containing both rectangular and polar coordinate pairs.

3. Colors

3.1 *Setting the Default Colors.*

`\drawcolor[`⟨*model*⟩`]{`⟨*colorspec*⟩`}`
`\fillcolor`...
`\hatchcolor`...
`\headcolor`...
`\tlabelcolor`...
`\backgroundcolor`...

These macros set the default color (METAPOST only) for various drawing elements. Any curve (with one exception, those drawn by `\plotdata`), whether solid, dashed, dotted, or plotted in symbols, will be in the color set by `\drawcolor`. Set the color used by `\gfill` with `\fillcolor`. For all the hatching commands use `\hatchcolor`, and for arrowheads, `\headcolor`. When mplabels is in effect, the color of labels can be set with `\tlabelcolor`, and one can set the color used by `\gclear` with `\backgroundcolor` (the same color is used in the interior of unfilled points drawn with `\point`). The optional ⟨*model*⟩ may be one of rgb, RGB, cmyk, gray, and named. The ⟨*colorspec*⟩ depends on the model, as outlined below. Each of these commands sets a corresponding METAPOST color variable with the same name (except `\backgroundcolor` sets the color `background`). Thus one can set the filling color to the drawing color with `\fillcolor{drawcolor}`.

3.2 METAPOST *Colors.*

If the optional ⟨*model*⟩ specification is omitted, the color specification may be any expression recognized as a color by METAPOST. In METAPOST, a color is a triple of numbers like `(1,.5,.5)`, with the coordinates between 0 and 1, representing red, green and blue levels, respectively. White is given by `(1,1,1)` and black by `(0,0,0)`. METAPOST also has color variables and several have been predefined: `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`, `white`, and `black`. All the names in the LaTeX COLOR package's `dvipsnam.def` are predefined color variable names. Since METAPOST allows color expressions, colors may be added and multiplied by numerics. Moreover, several color functions have been defined:

`cmyk(`c`,`m`,`y`,`k`)`

Converts a cmyk color specification to METAPOST's native rgb. For example, the command `cmyk(1,0,0,0)` yields `(0,1,1)`, which is the definition of `cyan`.

`RGB(`R`,`G`,`B`)`

Converts an RGB color specification to rgb. It essentially just divides each component by 255.

`gray(`g`)`

Converts a gray level to a multiple of `(1,1,1)`.

`named(`⟨*name*⟩`)`, `rgb(`*r*`,`*g*`,`*b*`)`

These are essentially no-ops. However; `rgb` will truncate the arguments to the 0–1 range, an unknown ⟨*name*⟩ is converted to `black`, and an unknown numeric argument is set to 0.

As an example of the use of these functions, one could conceivable write:

```
\drawcolor{0.5*RGB(255,0,0)+0.5*cmyk(1,0,0,0)}
```

to have all curves drawn in a color halfway between red and cyan (which turns out to be a dark gray).

3.3 *Color Models.*

When the optional ⟨*model*⟩ is specified in the color setting commands, it determines the format of the color specification:

Model:	*Specification:*
rgb	Three numbers in the range 0 to 1 separated by commas.
RGB	Three numbers in the range 0 to 255 separated by commas.
cmyk	Four numbers in the range 0 to 1 separated by commas.
gray	One number in the range 0 to 1, with 1 indicating white, 0 black.
named	A METAPOST color variable name either predefined by MFPIC or by the user.

Example: `\fillcolor[cmyk]{1,.3,0,.2}` and `\fillcolor{cmyk(1,.3,0,.2)}` are essentially equivalent. Note that when the optional model is specified, the color specification must not be enclosed in parentheses. Note also that each model name is the name of a color function described in the previous subsection. That is how the models are implemented internally.

3.4 *Defining a Color Name.*

`\mfpdefinecolor{`⟨*name*⟩`}{`⟨*model*⟩`}{`⟨*colorspec*⟩`}`

This defines a color variable ⟨*name*⟩ for later use, either in the commands `\drawcolor`, etc., or in the optional parameters to `\draw`, etc. The name can be used alone or in the `named` model. The mandatory ⟨*model*⟩ and ⟨*colorspec*⟩ are as above.

A final caution, the colors of an MFPIC figure are stored in the `.mp` output file, and are not related to colors used or defined by the LaTeX COLOR package. In particular a color defined only by LaTeX's `\definecolor` command will remain unknown to MFPIC. Conversely, LaTeX commands will not recognize any color defined only by `\mfpdefinecolor`.

4. Shape-Modifier Macros.

Some MFPIC macros operate as *shape-modifier* macros—for example, if you want to put an arrowhead on a line segment, you could write: `\arrow\lines{(0,0),(1,0)}`. These are always prefixed to some figure drawing command, and apply only to the next following figure macro (which can be rather far removed) provided that only other prefix commands intervene. This is a rather long section, but even more modification prefixes are documented in subsection *Transformation of Paths* of section Affine Transforms.

In the PostScript context (i.e., METAPOST) it is important to note that each prefix modifies the result of the entire following sequence. In essence prefixes can be viewed as being applied in the opposite order to their occurrence. Example:

```
\dashed\gfill\rect{(0,0),(1,1)}
```

This adds the dashed outline to the filled rectangle. That is, first the rectangle is defined, then it is filled, then the outline is drawn in dashed lines. This makes a difference when colors other than black are used. Drawing is done with the center of the virtual pen stroked down the middle of the boundary, so half of its width falls inside the rectangle. On the other hand, filling is done right up to the boundary. In this example, the dashed lines are drawn on top of part of the fill. In the reverse order, the fill would cover part of the outline.

For the purposes of these macros, a distinction must be made in the figure macros between "open" and "closed" paths. A path that merely returns to its starting point is *not* automatically closed; such a path is open, and must be explicitly closed, for example by `\lclosed` (see below). (On the METAFONT level, path closure is achieved by some variant of `..cycle`). The (already) closed paths are: `\rect`, `\circle`, `\ellipse`, `\sector`, `\cyclic`, `\polygon`, `\plrregion`, `\chartbar`, `\piewedge`, `\tlabelrect`, `\tlabeloval`, `\tlabelellipse`, `\tlabelcircle` and `\btwnfcn` (below).

4.1 *Closure of Paths.*

`\lclosed...`

Makes each open path into a closed path by adding a line segment between the endpoints of the path.

`\bclosed...`
`\cbclosed...`

These macros are similar to `\lclosed`, except that they close each open path by drawing a Bézier, or a cubic B-spline, respectively, between the path's endpoints.

`\sclosed...`
`\uclosed...`

These allow METAFONT to choose its notion of the best smooth curve. The former allows the entire new curve to be redrawn. `\sclosed\curve` is more or less equivalent to `\cyclic` and closing a curve can cause METAFONT to rethink how the original was drawn. The command `\uclosed` allows METAFONT to select the closure (subject to smoothness), but prevents it from changing the original curve.

4.2 *Reversal and Connection of Paths.*

`\reverse`...

Turns a path around, reversing its orientation. This will affect both the direction of arrows (e.g. bi-directional arrows can be coded with `\arrow\reverse\arrow`... —here the first `\arrow` modifier applies to the *reversed* path), and the order of endpoints for a `\connect`...`\endconnect` environment (below).

`\connect` ... `\endconnect`

This pair of macros, acting as an environment, add line segments from the trailing endpoint of one open path to the leading endpoint of the next path, in the given order. The result is a connected, *open* path.

Note: In LaTeX, this pair of macros can be used in the form of a LaTeX-style environment called `connect` —as in `\begin{connect}`...`\end{connect}`.

4.3 *Drawing.*

When MFPIC is loaded, the initial way in which figures are drawn is with a solid outline. That is, `\lines{(1,0),(1,1),(0,0)}` will draw two solid lines connecting the points. When the macros in this section are used, any previously established default (see *Changing the Default Rendering* of section SHAPE-MODIFIER MACROS) is overridden.

`\draw[`⟨*color*⟩`]`...

Draws the subsequent path using a solid outline. For an example: to both draw a curve and hatch its interior, `\draw\hatch` must be used. Default for ⟨*color*⟩ is `drawcolor`.

`\dashed[`⟨*length*⟩`,`⟨*space*⟩`]`...

Draws dashed segments along the path specified in the next command. The default length of the dashes is the value of the TeX dimension `\dashlen`, initially `4pt`. The default space between the dashes is the value of the TeX dimension `\dashspace`, initially `4pt`. The dashes and the spaces between may be increased or decreased by as much as $\frac{1}{n}$ of their value, where n is the number of spaces appearing in the curve, in order to have the proper dashes at the ends. The dashes at the ends are half of `\dashlen` long.

`\dotted[`⟨*size*⟩`,`⟨*space*⟩`]`...

Draws dots along the specified path. The default size of the dots is the value of the TeX dimension `\dotsize`, initially `0.5pt`. The default space between the dots is the value of the TeX dimension `\dotspace`, initially `3pt`. The size of the spaces may be adjusted as in `\dashed`.

`\plot[`⟨*size*⟩`,`⟨*space*⟩`]{`⟨*symbol*⟩`}`...

Similar to `\dotted` except copies of ⟨*symbol*⟩ are drawn along the path. Possible symbols are those listed under `\plotsymbol` above. The default ⟨*size*⟩ is `\pointsize` and the default ⟨*space*⟩ is `\symbolspace`, initially `5pt`.

`\plotnodes[`⟨*size*⟩`]{`⟨*symbol*⟩`}`

This places a symbol (same possibilities as in `\plotsymbol`) at each node of the path that follows. A node is one of the points through which METAFONT draws its curve. If one of the macros `\polyline{...}` or `\curve{...}` follows, each of the points listed is a node. In the `\datafile` command (below), each of the data points in the file is. In the function macros (below) the points corresponding to ⟨*min*⟩, ⟨*max*⟩ and each step in between are nodes. The optional ⟨*size*⟩ defaults to `\pointsize`.

`\dashpattern{`⟨*name*⟩`}{`⟨*len1*⟩`,`⟨*len2*⟩`,...,`⟨*len2k*⟩`}`

For more general dash patterns than `\dashed` and `\dotted` provide, there is a generalized dashing command. One must first establish a named dashing pattern with this command. ⟨*name*⟩ should be a legal METAFONT variable name. (Any sequence of letters and underscores will work.) Try to make it distinctive to avoid undoing some internal variable. ⟨*len1*⟩ through ⟨*len2k*⟩ are an even number of lengths. The odd ones determine the lengths of dashes, the even ones the lengths of spaces. A dash of length `0pt` means a dot. An alternating dot-dash pattern can be specified with

`\dashpattern{dotdash}{0pt,4pt,3pt,4pt}`.

Note: Since pens have some thickness, dashes look a little longer, and spaces a little shorter, than the numbers suggest. It seems to my eyes better if one increases the specified spaces (and decreases the specified dashes) by the thickness of the drawing pen (normally `0.5pt`).

`\gendashed{`⟨*name*⟩`}...`

Once a dashing pattern name has been defined, it can be used in this command to draw the curve that follows it. Using a name not previously assigned will generate a METAFONT error, but TEX will not complain. If all the dimensions in a dashpattern are 0, `\gendashed` responds by drawing a solid curve. The same is true if the pattern has only one entry. If the pattern has an odd number of entries, the last one is ignored.

4.4 *Arrows.*

`\arrow[l`⟨*headlen*⟩`][r`⟨*rotate*⟩`][b`⟨*backset*⟩`][c`⟨*color*⟩`]...`

Draws an arrowhead at the endpoint of the open path (or at the last key point of the closed path) that follows. The optional parameter ⟨*headlen*⟩ determines the length of the arrowhead. The default is the value of the TEX dimension `\headlen`, initially `3pt`. The optional parameter ⟨*rotate*⟩ allows the arrowhead to be rotated counterclockwise around its point an angle of ⟨*rotate*⟩ degrees. The default is 0. The optional parameter ⟨*backset*⟩ allows the arrowhead to be "set back" from its original point, thus allowing e.g. double arrowheads. This parameter is in the form of a TEX dimension—its default value is `0pt`. If an arrowhead is both rotated and set back, the rotation affects the direction in which the arrowhead is set back. The optional ⟨*color*⟩ defaults to `headcolor`. The optional parameters may appear in any order, but the indicated key character for each parameter must always appear.

4.5 *Shading, Filling, Erasing, Clipping, Hatching.*

These macros can all be used to fill (or unfill) the interior of closed paths, even if the paths cross themselves. Filling an open curve is technically an error, but the code in `grafbase` responds by drawing the path and not doing any filling. These macros replace the default rendering, that is, when they are used the outline will not be drawn unless an explicit prefix to do so is present.

`\gfill[`⟨*color*⟩`]`...

Fills in the subsequent closed path. Under METAPOST it fills with ⟨*color*⟩, which defaults to `fillcolor`.

`\gclear`...

Erases everything *inside* the subsequent closed path. Under METAPOST it actually fills with the predefined color `background`. Since `background` is normally white and so are most actual backgrounds, this is usually indistinguishable from clearing.

`\gclip`...

Erases everything *outside* the subsequent closed path from the picture. Both `\gclear` and `\gclip` will affect previously placed tlabels if mplabels is in effect.

`\shade[`⟨*shadesp*⟩`]`...

Shades the interior of the subsequent closed path with dots. The diameter of the dots is the METAFONT variable `shadewd`, set by the macro `\shadewd{`⟨*size*⟩`}`. Normally this is `0.5pt`. The optional argument specifies the spacing between (the centers of) the dots, which defaults to the TEX dimension `\shadespace`, initially `1pt`. If `\shadespace` is less than `shadewd`, the closed path is filled with black, as if with `\gfill`. Under METAPOST this macro actually fills the path's interior with a shade of gray. The shade to use is computed based on `\shadespace` and `shadewd`. The default values of these parameters correspond to a gray level of 75% of white.* The METAFONT version attempts to optimize the dots to the pixel grid corresponding to the printers resolution (to avoid generating "dither lines"). Because this involves rounding, it will happen that values of `\shadespace` that are relatively close and at the same time close to `shadewd` produce exactly the same shade. By bad luck, when `shadewd` is the default `0.5pt`, and the printer resolution is 300dpi, values of `\shadespace` between `.69pt` and `1pt` create the same shading pattern. Most of the time, however, values which differ by at least 20% will produce different patterns.

`\polkadot[`⟨*space*⟩`]`...

Fills the interior of a closed path with large dots. This is almost what `\shade` does, but there are several differences. `\shade` is intended solely to simulate a gray fill in METAFONT where the only color is black. So it is optimized for small dots aligned to the pixel

* If `\shadewd` is w and `\shadespace` is s, then the level of gray is $1-(w/s)^2$, where 0 denotes black and 1 white.

grid (in METAFONT). In METAPOST all it does is fill with gray and is intended merely for compatibility. The macro `\polkadot` is intended for large dots in any color, and so it optimizes spacing (a nice hexagonal array) and makes no attempt to align at the pixel level. The ⟨*space*⟩ defaults to the TEX dimension `\polkadotspace`, initially `10pt`. The diameter of the dots is the value of the METAFONT variable `polkadotwd`, which can be set with `\polkadotwd{`⟨*size*⟩`}`, and is initially `5pt`. The dots are colored with `fillcolor`.

`\thatch[`⟨*hatchsp*⟩`,`⟨*angle*⟩`] [`⟨*color*⟩`]`...

Fills a closed path with equally spaced parallel lines at the specified angle. The thickness of the lines is set by the macro `\hatchwd`. In the optional argument, ⟨*hatchsp*⟩ specifies the space between lines, which defaults to the TEX dimension `\hatchspace`, initially `3pt`. The ⟨*angle*⟩ defaults to 0. The ⟨*color*⟩ defaults to `hatchcolor`. If `\hatchspace` is `0pt` (or less), the closed path is filled with ⟨*color*⟩, as if with `\gfill`. Of the first optional arguments, either both must be present, or both must be absent. For the color argument to be present, the other optional arguments must also be present.

`\lhatch[`⟨*hatchsp*⟩`] [`⟨*color*⟩`]`...

Draws lines shading in the subsequent closed path in a "left-oblique hatched" (upper left to lower right) pattern. It is exactly the same as `\thatch[`⟨*hatchsp*⟩`,-45] [`⟨*color*⟩`]`...

`\rhatch[`⟨*hatchsp*⟩`] [`⟨*color*⟩`]`...

Draws lines shading in the subsequent closed path in a "right-oblique hatched" (lower left to upper right) pattern. It is exactly the same as `\thatch[`⟨*hatchsp*⟩`,45] [`⟨*color*⟩`]`...

`\hatch[`⟨*hatchsp*⟩`] [`⟨*color*⟩`]`...
`\xhatch[`⟨*hatchsp*⟩`] [`⟨*color*⟩`]`...

Draws lines shading in the subsequent closed path in a "cross-hatched" pattern. It is exactly the same as `\rhatch` followed by `\lhatch` using the same ⟨*hatchsp*⟩ and ⟨*color*⟩.

4.6 *Changing the Default Rendering.*

Rendering is the process of converting a geometric description into a drawing. In METAFONT, this means producing a bitmap (METAFONT calls this a `picture`), either by stroking (drawing) a path using a particular pen), or by filling a closed path. In METAPOST it means producing a POSTSCRIPT description of strokes with pens, and fills

`\setrender{`⟨TEX *commands*⟩`}`

Initially, MFPIC uses the `\draw` command (stroking) as the default operation when a figure is to be rendered. However, this can be changed to any combination of MFPIC rendering commands and/or other TEX commands, by using the `\setrender` command. This is a local redefinition, so it can be enclosed in braces to restrict its range.

For example, after `\setrender{\dashed\shade}` the command `\circle{(0,0),1}` produces a shaded circle with a dashed outline. Any explicit drawing or filling prefix replaces this default. Other kinds of prefixes (e.g., `\lclosed`, `\arrow`, `\rotatepath`) do not.

4.7 *Examples.*

It may be instructive, for the purpose of seeing how the syntax of *shape-modifier switches* works, to consider two examples:

```
\draw\shade\lclosed\lines{...}
```

which shades inside a polygon and draws its outline; and

```
\shade\lclosed\draw\lines{...}
```

which draws all of the outline *except* the line segment supplied by `\lclosed`, then shades the interior. And in METAPOST this is just a gray fill which covers the inner half of the stroke made by `\draw`.

5. Functions and Plotting.

In the following macros, expressions like $f(\texttt{x})$, $g(\texttt{t})$ stand for any legal METAFONT expression, in which the only unknown variables are those indicated (`x` in the first case, and `t` in the second).

5.1 *Defining Functions*

`\fdef{`⟨*fcn*⟩`}{`⟨*param1*⟩`,`⟨*param2*⟩`,...}{`⟨*mf-expr*⟩`}`

Defines a METAFONT function ⟨*fcn*⟩ of the parameters ⟨*param1*⟩, ⟨*param2*⟩, ..., by the METAFONT expression ⟨*mf-expr*⟩ in which the only free parameters are those named. The return type of the function is the same as the type of the expression. The function name ⟨*fcn*⟩ is subject to METAFONT's rule for variable names. Try to make the names distinctive to avoid redefining internal METAFONT commands.

The expression ⟨*mf-expr*⟩ is passed directly into the corresponding METAFONT macro and interpreted there, so METAFONT's rules for algebraic expressions apply.

As an example, after `\fdef{myfcn}{s,t}{s*t-t}`, any place below where a METAFONT expression is required, you can use `myfcn(2,3)` to mean `2*3-3` and `myfcn(x,x)` to mean `x*x-x`.

Operations available include `+`, `-`, `*`, `/`, and `**` (`x**y`$= x^y$), with `(` and `)` for grouping. Functions already available include the standard METAFONT functions `round`, `floor`, `ceiling`, `abs`, `sqrt`, `sind`, `cosd`, `mlog`, and `mexp`. Note that in METAFONT the operations `*` and `**` have the same level of precedence, so `x*y**z` means $(xy)^z$. Use parentheses liberally!

(*Notes:* The METAFONT trigonometric functions `sind` and `cosd` take arguments in degrees; `mlog(x)`$= 256 \ln x$, and `mexp` is its inverse.) You can also define the function ⟨*fcn*⟩ by cases, using the METAFONT conditional expression

`if` ⟨*boolean*⟩`:` ⟨*expr*⟩ `elseif` ⟨*boolean*⟩`:` ... `else:` ⟨*expr*⟩ `fi`.

Relations available for the ⟨*boolean*⟩ part of the expression include `=`, `<`, `>`, `<=`, `<>` and `>=`.

Complicated functions can be defined by a compound expression, which is a series of METAFONT statements, followed by an expression, all enclosed in the METAFONT commands `begingroup` and `endgroup`. METAFONT functions can call METAFONT functions, recursively.

Many common functions have been predefined in `grafbase`. These include all the usual trig functions `tand`, `cotd`, `secd`, `cscd`, which take angles in degrees, plus variants `sin`, `cos`, `tan`, `cot`, `sec`, and `csc`, which take angles in radians. Some inverse trig functions are also available, the following produce angles in degrees: `asin`, `acos`, and `atan`, and the following in radians: `invsin`, `invcos`, `invtan`. The exponential and hyperbolic functions: `exp`, `sinh`, `cosh`, `tanh`, and their inverses `ln` (or `log`), `asinh`, `acosh`, and `atanh` are also defined.

5.2 *Plotting Functions*

The plotting macros take two or more arguments. They have an optional first argument, ⟨*spec*⟩, which determines whether a function is drawn smooth (as a METAFONT Bézier curve), or polygonal (as line segments)—if ⟨*spec*⟩ is `p`, the function will be polygonal. Otherwise the ⟨*spec*⟩ should be `s`, followed by an optional positive number no smaller than 0.75. In this case the function will be smooth with a tension equal to the number. See the `\curve` command for an explanation of tension. The default ⟨*spec*⟩ depends on the purpose of the macro.

One compulsory argument contains three values ⟨*min*⟩, ⟨*max*⟩ and ⟨*step*⟩ separated by commas. The independent variable of a function starts at the value ⟨*min*⟩ and steps by ⟨*step*⟩ until reaching ⟨*max*⟩. If ⟨*max*⟩ − ⟨*min*⟩ is not a whole number of steps, then round((⟨*max*⟩ − ⟨*min*⟩)/⟨*step*⟩) equal steps are used. One may have to experiment with the size of ⟨*step*⟩, since METAFONT merely connects the points corresponding to these steps with what *it* considers to be a smooth curve. Smaller ⟨*step*⟩ gives better accuracy, but too small may cause the curve to exceed METAFONT's capacity or slow down its processing. Increasing the tension may help keep the curve in line, but at the expense of reduced smoothness.

There are one or more subsequent arguments, each of which is a METAFONT function or expression as described above.

`\function[`⟨*spec*⟩`]{`⟨*xmin*⟩`,`⟨*xmax*⟩`,`⟨*step*⟩`}{`*f*`(x)}`

Plots *f*(`x`), a METAFONT numeric function or expression of one numeric argument, which must be denoted by a literal `x`. The default ⟨*spec*⟩ is `s`.

`\parafcn[`⟨*spec*⟩`]{`⟨*tmin*⟩`,`⟨*tmax*⟩`,`⟨*step*⟩`}{`⟨*pfcn*⟩`}`

Plots the parametric path determined by $(x(t), y(t)) =$⟨*pfcn*⟩(`t`), where ⟨*pfcn*⟩ is a METAFONT function or expression of one numeric argument `t`, returning a METAFONT *pair* (such as (x, y)). Or a pair of numeric expressions enclosed in parentheses and separated by a comma. The default ⟨*spec*⟩ is `s`. Example: `\parafcn{ 0,1,.1}{2*(t,t*t)}` plots a smooth parabola from $(0, 0)$ to $(2, 2)$.

`\plrfcn[`⟨*spec*⟩`]{`⟨θ*min*⟩`,`⟨θ*max*⟩`,`⟨θ*step*⟩`}{`*f*`(t)}`

Plots the polar function determined by $r = f(\theta)$, where f is a METAFONT numeric function or expression of one numeric argument, and θ varies from ⟨θ*min*⟩ to ⟨θ*max*⟩ in steps of ⟨θ*step*⟩. Each θ value is interpreted as an angle measured in *degrees*. In the expression *f*(`t`), the unknown `t` stands for θ. The default ⟨*spec*⟩ is `s`.

`\btwnfcn[`⟨*spec*⟩`]{`⟨*xmin*⟩`,`⟨*xmax*⟩`,`⟨*step*⟩`}{`f`(x)}{`g`(x)}`

Draws the region between the two functions $f(x)$ and $g(x)$, these being numeric functions of one numeric argument `x`. The region is bounded also by the vertical lines at ⟨*xmin*⟩ and ⟨*xmax*⟩. Unlike the previous function macros, the default ⟨*spec*⟩ is `p` —this macro is intended to be used for shading between drawn functions, a task for which smoothness is usually unnecessary.

`\plrregion[`⟨*spec*⟩`]{`⟨θ*min*⟩`,`⟨θ*max*⟩`,`⟨θ*step*⟩`}{`f`(t)}`

Plots the polar region determined by $r = f(\theta)$, where f is a METAFONT numeric function of one numeric argument `t`. The θ values are angles (measured in *degrees*), varying from ⟨θ*min*⟩ to ⟨θ*max*⟩ in steps of ⟨θ*step*⟩. In the expression f(`t`), the `t` stands for θ. The region is also bounded by the angles ⟨θ*min*⟩ and ⟨θ*max*⟩, i.e. by the line segments joining the origin to the endpoints of the function. The default ⟨*spec*⟩ is `p` —this macro is intended to be used for shading a region with the boundary drawn, a task for which smoothness is usually unnecessary.

5.3 *Plotting external data files*

`\datafile[`⟨*spec*⟩`]{`⟨*file*⟩`}`
`\smoothdata[`⟨*tension*⟩`]`
`\unsmoothdata`

`\datafile` defines a curve connecting the points listed in the file ⟨*file*⟩. The ⟨*spec*⟩ may be either `p` or `s` followed by a tension value (as in `\curve`). If no ⟨*spec*⟩ is given, the default is initially `p`, but `\smoothdata` may be used to change this. That is, after `\smoothdata[`⟨*tension*⟩`]` the default ⟨*spec*⟩ is changed to `s`⟨*tension*⟩. If the tension parameter is not supplied it defaults to `1.0`. The command `\unsmoothdata` restores the default ⟨*spec*⟩ to `p`.

By default, each non-blank line in the file is assumed to contain at least two numbers, separated by whitespace (blanks or tabs). The first two numbers on each line are assumed to represent the x- and y-coordinates of a point. Initial blank lines in the file are ignored, as are comments. The comment character in the data file is assumed to be `%`, but it can be reset using `\mfpdatacomment` (below). Any blank line other than at the start of the file causes the curve to terminate. The `\datafile` command may be preceded by any of the prefix commands, so that, for example, a closed curve could be formed with `\lclosed\datafile{data.dat}`.

The `\datafile` command has a second, independent use. Any MFPIC command (other than one that prints text labels) that takes as its last argument a list of points (or numerical values), separated by commas, can now have that list replaced by values in an external data file. For example, if a file `numlist.dat` contains one or more numerical value per line separated by blanks, then

```
\using{#1 #2}{#1}
\xmarks\datafile{numlist.dat}
```

can be used to put hash marks on the x-axis at each of the points corresponding to the first number on each line. This `\using` command tells `\mfpic` to discard all but the

first number on each line and include that number in the list. The MFPIC macros that allow this usage of `\datafile` are, for numeric data: `\piechart`, `\barchart` , and all the axis marks commands; for point or vector data: `\point`, `\plotsymbol`, `\polyline`, `\polygon`, `\fcncurve`, `\curve`, `\cyclic`, `\turtle`, `\qspline`, `\closedqspline`, `\cspline`, and `\closedcspline`.

The `\datafile` drawing command described above produces the equivalent of using `\polyline\datafile` or of `\curve[`⟨*tension*⟩`]\datafile` if `\smoothdata[`⟨*tension*⟩`]` has been used.

`\mfpdatacomment\`⟨*char*⟩

Changes ⟨*char*⟩ to a comment character and changes the usual TEX comment character `%` to an ordinary character *while reading a datafile for drawing.*

`\using{`⟨*in-pattern*⟩`}{`⟨*out-pattern*⟩`}`

Used to change the assumptions about the format of the data file. For example, if there are four numbers on each line separated by commas, to plot the third against the second (in that order) you can say `\using{#1,#2,#3,#4}{(#3,#2)}`. This means the following: Everything on a line up to the first comma is assigned to parameter `#1`, everything from there up to the second comma is assigned to parameter `#2`, etc. Everything from the third comma to the end of line is assigned to `#4`. When the line is processed by TEX a METAFONT pair is produced representing a point on the curve. METAFONT pair expressions can be used in the output portion of `\using`. For example `\using{#1,#2,#3}{(#2,#1)/10}` or even `\using{#1,#2,#3}{polar(#2,#1)}` if the data are polar coordinates. As a special case, you can plot any number against its sequence position, with something like `\using{#1,#2,#3}{(\sequence,#1)}`. Here, the macro `\sequence` will take on the values 1, 2, etc. as lines are read from the file. The default setting is `\using{#1 #2 #3}{(#1,#2)}`. MFPIC provides the command `\usingpairdefault` to reset this default. It also provides the command `\usingnumericdefault` as the equivalent of `\using{#1 #2}{#1}` (seen above in the `\xmarks\datafile` example). The `\using` command cannot normally be used in the replacement text of another command.

Note that the default value of `\using` appears to reference three arguments. If there are only two numbers on a line separated by whitespace, this will still work because of TEX's argument matching rules. TEX's file reading mechanism normally converts the EOL to a space, but there are exceptions so MFPIC always adds a space at the end of each line read in to be on the safe side. Then the default definition of `\using` reads everything up to the first space as `#1` (whitespace is normally compressed to a single space by TEX's reading mechanism), then everything to the second space (the one added at the end of the line, perhaps) is `#2`, then everything to the EOL is `#3`. This might assign an empty argument to `#3`, but it is discarded anyway.

If the numerical data contain percentages with explicit `%` signs, then choose another comment character with `\mfpdatacomment`. This will change `%` to an ordinary character *in the data file.* However, in your `\using` command it would still be read as a comment. The following example shows how to overcome this:

```
\makepercentother
\using{#1% #2 #3}{(#1/100,#2)}
\makepercentcomment
```

Here is an analysis of the meaning of this example: everything in a line, up to the first percent followed by a space is assigned to parameter `#1`, everything from there to the next space is assigned to `#2` and the rest of the line (which may be empty) is `#3`. On the output side in the above example, the percentage is divided by 100 to convert it to a fraction, and plotted against the second parameter. Note: normal comments should not be used between `\makepercentother` and `\makepercentcomment`, for obvious reasons.

`\plotdata[`⟨*spec*⟩`]{`⟨*file*⟩`}`

This plots several curves from a single file. The ⟨*spec*⟩ and the command `\smoothdata` have the same effect on each curve as in the `\datafile` command. The data for each curve is a succession of nonblank lines separated from the data for the next curve by a single blank line. A *pair* of successive blank lines is treated as the end of the data. No prefix macros are permitted in front of `\plotdata`.

Each successive curve in the data file is drawn differently. By default, the first is drawn as a solid line the next dashed, the third dotted, etc., through a total of six different line types. A `\gendashed` command is used with predefined dashpatterns named `dashtype0` through `dashtype5`. This behavior can be changed with:

```
\coloredlines
\pointedlines
\datapointsonly
\dashedlines
```

The command `\coloredlines` changes to cycling through eight different colors (starting with black). This has an effect only for METAPOST. The command `\pointedlines` causes `\plotdata` to use `\plot{`⟨*symbol*⟩`}` commands, cycling through nine different symbols. The command `\datapointsonly` causes `\plotdata` to use `\plotnodes{`⟨*symbol*⟩`}` commands to plot the data points only. (See the **Appendix** for more details.) The command `\dashedlines` restores the default. If, for some reason, you do not like the default starting line style (say you want to start with a color other than black), you can use one of the following commands.

`\mfplinetype{`⟨*num*⟩`}`, or
`\mfplinestyle{`⟨*num*⟩`}`

Here ⟨*num*⟩ is a non-negative number, less than the number of different drawing types available. The four previous commands reset the number to 0, so if you use one of them, issue `\mfplinetype` *after* it. The different line styles are numbered starting from 0. If two or more `\plotdata` commands are used in the same `mfpic` environment, the numbering in each continues where the one before left off (unless you issue one of the commands above in between). `\mfplinestyle` means the same as `\mfplinetype`, and is included for compatibility. See the **Appendix** to find out what dashpattern, color or symbol corresponds to each number.

The sole exception to the general rule that all curves are drawn in `drawcolor` is the `\plotdata` command after `\coloredlines` has been issued.

The commands `\using`, `\mfpdatacomment` and `\sequence` have the same meaning for `\plotdata` as they do for `\datafile` (above). The sequence numbering for `\sequence` starts over with each new curve.

6. Labels and Captions.

6.1 *Setting Text.*

The macros `\tlabel`, `\tlabels`, `\axislabels` and `\tcaption` do not affect the METAFONT file (⟨*file*⟩`.mf`) at all, but are added to the picture by TEX. If metapost is in effect but mplabels is not, they do not affect the METAPOST file. In these cases, if these macros are the only changes or additions to your document, there is no need to repeat the processing with METAFONT or METAPOST nor the reprocessing with TEX in order to complete your TEX document.

`\tlabel[`⟨*just*⟩`](`⟨x⟩`,`⟨y⟩`){`⟨*label text*⟩`}`
`\tlabels{`⟨*params*$_1$⟩ ⟨*params*$_2$⟩ ...`}`

Places TEX labels on the graph. (Not to be confused with LATEX's `\label` command.) The special form `\tlabels` (note the plural) essentially just applies `\tlabel` to each set of parameters listed in its argument. That is, each ⟨*params*$_k$⟩ is a valid set of parameters for a `\tlabel` command. These can be separated by spaces, newlines, or nothing at all.

The last required parameter is ordinary TEX text. The pair `(`⟨x⟩`,`⟨y⟩`)` gives the coordinates of a point in the graph where the text will be placed. It may optionally be enclosed in braces. In fact, if mplabels is in effect the syntax

`\tlabel[`⟨*just*⟩`]{`⟨*pair-list*⟩`}{`⟨*label text*⟩`}`

may be used, where ⟨*pair-list*⟩ is any expression recognized as a pair by METAPOST, or a comma-separated list of such pairs.

The optional parameter `[`⟨*just*⟩`]` specifies the *justification*, the relative placement of the label with respect to the point `(`⟨x⟩`,`⟨y⟩`)`. It is a two-character sequence where the first character is one of `t` (top), `c` (center), `b` (bottom), or `B` (Baseline), to specify vertical placement, and the second character is one of `l` (left), `c` (center), or `r` (right), to specify horizontal placement. These letters specify what part of the *text* is to be placed at the given point, so `r` puts the right end of the text there—which means the text will be left of the point.

When mplabels is in effect, the two characters may optionally be followed by a number, specifying an angle in degrees to rotate the text about the point `(`⟨x⟩`,`⟨y⟩`)`. If the angle is supplied without mplabels it is ignored after a warning. If the angle is absent, there is no rotation. Note that the rotation takes place after the placement. For example, `[cr]` will place the text left of the point, while `[cr180]` will place the text right of the point (and upsidedown, of course).

There should be no spaces before or between the first two characters. However the number, if present, is only required to be a valid METAPOST numerical expression containing

no bracket characters; as such, it may contain some spaces (e.g., around operations as in `45 + 30`).

A multiline `\tlabel` may be specified by explicit line breaks, which are indicated by the `\\` command. This is a very rudimentary feature. By default it left justifies the lines and causes `\tlabel` to redefine `\\`. One can center a line by putting `\hfil` as the first thing in the label text, and right justify by putting `\hfill` there (these are TEX primitives). Redefining `\\` can interfere with LATEX's definition. For better control in LATEX use `\shortstack` inside the label (or a `tabular` environment or some other environment which always initializes `\\` with its own definition).

If the label goes beyond the bounds of the graph in any direction, the box containing the graph is expanded to make room for it. (Note: in this behavior it differs from the LATEX `picture` environment.)

If the mplabels option is in effect, `\tlabel` will write a `btex ... etex` group to the output file, allowing METAPOST to arrange for typesetting the label. In this case, the label is part of the picture, rather than being laid on top of it. It may therefore be covered up by MFPIC later filling macros, or clipped off by `\gclear` or`\gclip`.

`\everytlabel{`⟨TEX-*code*⟩`}`

One problem with multi-line `\tlabel`s is that each line of their contents constitutes a separate group. This makes it difficult to change the `\baselineskip` (for example) inside a label. The command `\everytlabel` saves it's contents in a token register and the code is issued in each `\tlabel`, as the last thing before the actual line(s) of text. Any switch you want to apply to every line can be supplied. For example

```
\everytlabel{\bf\baselineskip 10pt}
```

will make every line of every `\tlabel`'s text come out bold with 10 point baselines. The effect of `\everytlabel` is local to the `mfpic` environment, if it is issued inside one. Note that the lines of a tlabel are wrapped in a box, but the commands of `\everytlabel` are outside all of them, so no actual text should be produced by these commands.

Using `\tlabel` without an optional argument is equivalent to specifying `[Bl]`. Use the following command to change this behavior.

`\tlabeljustify{`⟨*just*⟩`}`

After this command the placement of all subsequent labels without optional argument will be as specified in this command. For example, `\tlabeljustify{cr45}` would cause all subsequent `\tlabel` commands lacking an optional argument to be placed as if the argument `[cr45]` were used in each.

`\tlabeloffset{`⟨*hlen*⟩`}{`⟨*vlen*⟩`}`
`\tlabelsep{`⟨*len*⟩`}`

The first command causes all subsequent `\tlabel` commands to shift the label right by ⟨*hlen*⟩ and up by ⟨*vlen*⟩ (negative lengths cause it to be shifted left and down, respectively).

The `\tlabelsep` command causes labels to be shifted by the given amount in a direction that depends on the optional positioning parameter. For example, if the first letter is

`t` the label is shifted down by the amount ⟨*len*⟩ and if the second letter is `l` it is also shifted right. In all cases it is shifted *away* from the point of placement (unless the dimension is negative). If `c` or `B` is the first parameter, no vertical shift takes place, and if `c` is the second, there is no horizontal shift. This is intended to be used in cases where something has been drawn at that particular point, in order to separate the text from the drawing, but the value is also written to the output file for use by `\tlabelrect` and related commands.

`\axislabels{`⟨*axis*⟩`}[`⟨*just*⟩`]{{`⟨*txt*$_1$⟩`}`⟨n_1⟩`, {`⟨*txt*$_2$⟩`}`⟨n_2⟩`,...>}`

This command places the given TEX text (⟨*txt*$_k$⟩) at the given positions (⟨n_k⟩) on the given axis, ⟨*axis*⟩, which must be a single letter and one of `l`, `b`, `r`, `t`, `x`, or `y`. The text is placed as in `\tlabels` (including the taking into account of `\tlabelsep` and `\tlableoffset`), except that the default justification depends on the axis (the settings of `\tlabeljustify` are ignored). In the case of the border axes, the default is to place the label outside the axis and centered. So, for example, for the bottom axis it is `[tc]`. The defaults for the x- and y-axis are below and left, respectively. The optional ⟨*just*⟩ can be used to change this. For example, to place the labels *inside* the left border axis, use `[cl]`. If mplabels is in effect, rotations can be included in the justification parameter. For example, to place the text strings "first", "second" and "third" just below the positions 1, 2 and 3 on the x-axis, rotated so they read upwards at a 90 degree angle, one can use `\axislabels{x}[cr90]{{first}1, {second}2, {third}3}`

`\plottext[`⟨*just*⟩`]{`⟨*text*⟩`}{(`x_0`,`y_0`), (`x_1`,`y_1`), ...}`

Similar in effect to `\point` and `\plotsymbol` (but without requiring METAFONT), `\plottext` places a copy of ⟨*text*⟩ at each of the listed points. It simply issues multiple `\tlabel` commands with the same text and optional parameter, but at the different points listed. This is intended to plot a set of points with a single letter or font symbol (instead of a METAFONT generated shape). Like `\axislabels`, this does not respond to the setting of `\tlabeljustify`. It has a default setting of `[cc]` if the optional argument is omitted. The points may be MetaPost pair expressions enclosed in braces under mplabels. This command is actually unnecessary under mplabels as the plain `\tlabel` command can then be given a list of points. The `\tlabel` command is more efficient.

`\mfpverbtex{`⟨TEX-*cmds*⟩`}`

This writes a `verbatimtex` block to the `.mp` file. It makes sense only if the mplabels option is used and so only for METAPOST. The ⟨TEX-*cmds*⟩ in the argument are written to the `.mp` file, preceded by the METAPOST command `verbatimtex` and followed by `etex`. Line breaks within the ⟨TEX-*cmd*⟩ are preserved. The `\mfpverbtex` command must come before any `\tlabel` that is to be affected by it. Any settings common to all `mfpic` environments should be in a `\mfpverbtex` command preceding all such environments. It may be issued at any point after MFPIC is loaded, and any number of times. Because of the way METAPOST handles `verbatimtex` material, the effects are not constrained by any grouping unless one places grouping commands within `verbatimtex material`.

`\tcaption[`⟨*maxwd*⟩`,`⟨*linewd*⟩`]{`⟨*caption text*⟩`}`

Places a TEX caption at the bottom of the graph. (Not to be confused with LATEX's similar `\caption` command.) The macro will automatically break lines which are too much wider than the graph—if the `\tcaption` line exceeds ⟨*maxwd*⟩ times the width of the graph, then lines will be broken to form lines at most ⟨*linewd*⟩ times the width of the graph. The default settings for ⟨*maxwd*⟩ and ⟨*linewd*⟩ are 1.2 and 1.0, respectively. `\tcaption` typesets its argument twice (as does LATEX's `\caption`), the first time to test its width, the second time for real. Therefore, the user is advised *not* to include any global assignments in the caption text.

If the `\tcaption` and graph have different widths, the two are centered relative to each other. If the tcaption takes multiple lines, then the lines are both left- and right-justified (except for the last line), but the first line is not indented. If the option centeredcaptions is in effect, each line of the caption will be centered.

In a `\tcaption`, Explicit line breaks may be specified by using the `\\` command.

Many MFPIC users find the `\tcaption` command too limiting (one cannot, for example, place the caption to the side of the figure). It is common to use some other method (such as LATEX's `\caption` command in a `figure` environment). The dimensions `\mfpicheight` and `\mfpicwidth` (see section PARAMETERS below) might be a convenience for plain TEX users who want to roll their own caption macros.

6.2 *Curves Surrounding* `\tlabel`*s*

`\tlabelrect[`⟨*rad*⟩`](`⟨*x*⟩`,`⟨*y*⟩`){`⟨*text*⟩`}`
`\tlabelrect[`⟨*rad*⟩`]{`⟨*pair*⟩`}{`⟨*text*⟩`}`

Produces a closed rectangle containing the *text-box* of ⟨*text*⟩. The text-box is here defined to be the bounding box of the text, increased on all four sides by the value set with `\tlabelsep` (see the section LABELS AND CAPTIONS, above). This command places the text centered at (⟨*x*⟩,⟨*y*⟩). Under mplabels, a pair expression in braces may be used to specify the point. This command may be preceded by prefix macros (see the section SHAPE-MODIFIER MACROS, above). The optional argument ⟨*rad*⟩ is a dimension, defaulting to `0pt`, that produces rounded corners made from quarter-circles of the given radius. If the corners are rounded, the sides are expanded slightly so the resulting shape still encompasses the complete text-box. There are no provisions for rotating the text, even if mplabels is in effect, nor will it do any justification except centering. There is a "star form" `\tlabelrect*` which produces the rectangle but omits placing the text.

`\tlabeloval[`⟨*mult*⟩`](`⟨*x*⟩`,`⟨*y*⟩`){`⟨*text*⟩`}`
`\tlabeloval[`⟨*mult*⟩`]{`⟨*pair*⟩`}{`⟨*text*⟩`}`

Similar to `\tlabelrect`, except it draws an ellipse. The ellipse is calculated to have the same ratio of width to height as the *text-box* (defined above), the optional ⟨*mult*⟩ is a multiplier that increases or decreases this ratio. Values of ⟨*mult*⟩ larger than 1 increase the width and decrease the height. The ellipse is always sized so that it passes through the four corners of the text box. There are no provisions for rotating the text, even if mplabels

is in effect, nor will it do any justification except centering. As with `\tlabelrect`, there is a "star form" that omits the text.

`\tlabelellipse[`⟨*ratio*⟩`](`⟨*x*⟩`,`⟨*y*⟩`){`⟨*text*⟩`}`
`\tlabelellipse[`⟨*ratio*⟩`]{`⟨*pair*⟩`}{`⟨*text*⟩`}`
`\tlabelcircle(`⟨*x*⟩`,`⟨*y*⟩`){`⟨*text*⟩`}`
`\tlabelcircle{`⟨*pair*⟩`}{`⟨*text*⟩`}`

Draws the smallest ellipse centered at the point that encompasses the *text-box* (defined as above) and that has a ratio of width to height equal to ⟨*ratio*⟩; then places the given text centered at the point. The default ratio is 1, which produces a circle. `\tlabelcircle` is the same as `\tlabelellipse`, and is intended for use when no optional argument is given in order to describe the shape being drawn. As with `\tlabelrect`, there is a "star form" that omits the text.

In the above `\tlabel...` curves, the optional parameter should be positive. If it is zero, all the curves silently revert to `\tlabelrect`. If it is negative, it is silently accepted. In the case of `\tlabelrect` this causes the corners to be concave and reversed, a rather bizarre effect. In the other cases, there is no visible effect, except the sense of the curve is reversed.

One can surround rotated text with one of these curves by the simple expedient of using the transformation prefix `\rotatepath` (see the section TRANSFORMATION OF PATHS, below) with the star form, and then add the same text with a `\tlabel` command at the same point with the same rotation.

7. SAVING AND REUSING AN MFPIC PICTURE.

These commands have been changed from versions prior to 0.3.14 in order to behave more like the LaTeX's `\savebox`, and also to allow the reuse of an allocated box. Past files that use `\savepic` will have to be edited to add `\newsavepic` commands that allocate the TeX boxes.

`\newsavepic{`⟨*picname*⟩`}`
`\savepic{`⟨*picname*⟩`}`
`\usepic{`⟨*picname*⟩`}`

`\newsavepic` allocates a box (like LaTeX's `\newsavebox`) in which to save a picture. As in `\newsavebox`, ⟨*picname*⟩ is a control sequence. Example: `\newsavepic{\foo}`.

`\savepic` saves the *next* `\mfpic` picture in the named box, which should have been previously allocated with `\newsavepic`. (This command should not be used *inside* an `mfpic` environment.) The next picture will not be placed, but saved in the box for later use. This is primarily intended as a convenience. One *could* use `\savebox{`⟨*picname*⟩`}{`⟨*entire* `mfpic` *environment*⟩`}`, but `\savepic` avoids having to place the `mfpic` environment in braces, and avoids one extra level of TeX grouping. It also avoids reading the entire `mfpic` environment as a parameter, which would nullify MFPIC's efforts to preserve line breaks in parameters written to the METAFONT output file. If you repeat `\savepic` with the same ⟨*picname*⟩, the old contents are replaced with the next picture.

`\usepic` copies the picture that had been saved in the named box. This may be repeated as often as liked to create multiple copies of one picture.

8. Picture frames.

When TEX is run but before METAFONT or METAPOST has been run on the output file, MFPIC detects that the `.tfm` file is missing or that the first METAPOST figure file ⟨*file*⟩`.1` is missing. In these cases, the `mfpic` environment draws only a rectangular frame with dimensions equal to the nominal size of the picture, containing the figure name and number (and any TEX labels). The command(s) used internally to do this are made available to the user.

`\mfpframe[`⟨*fsep*⟩`]`⟨ *material-to-be-framed* ⟩`\endmfpframe`
`\mfpframed[`⟨*fsep*⟩`]{`⟨*material-to-be-framed*⟩`}`

These surround their contents with a rectangular frame consisting of lines with thickness `\mfpframethickness` separated from the contents by the ⟨*fsep*⟩ if specified, otherwise by `\mfpframesep`. The default value of the TEX dimensions `\mfpframesep` and `\mfpframethickness` are `2pt` and `0.4pt`, respectively. The `\mfpframe` ... `\endmfpframe` version is preferred around `mfpic` environments or verbatim material since it avoids reading the enclosed material before appropriate `\catcode` changes go into effect. In LATEX, one can also use the `\begin{mfpframe}` ... `\end{mfpframe}` syntax.

An alternative way to frame `mfpic` pictures is to save them with `\savepic` (see previous section) and issue a corresponding `\usepic` command inside any framing environment/command of the user's choice or devising.

9. Affine Transforms.

Coordinate transformations that keep parallel lines in parallel are called **affine transforms**. These include translation, rotation, reflection, scaling and skewing (slanting). For the METAFONT coordinate system only—that is, for paths, but not for `\tlabel`'s (let alone `\tcaption`'s)—MFPIC provides the ability to apply METAFONT affine transforms.

9.1 *Affine Transforms of the* METAFONT *Coordinate System.*

`\coords` ... `\endcoords`

All affine transforms are restricted to the innermost enclosing `\coords`...`\endcoords` pair. If there is *no* such enclosure, then the transforms will apply to the rest of the `mfpic` environment

Note: In LATEX, a `coords` environment may be used.

`\applyT{`⟨*transformer*⟩`}`

Apply the METAFONT ⟨*transformer*⟩ to the current coordinate system. For example, the MFPIC TEX macro `\zslant#1` is implemented as `\applyT{zslanted #1}` where the argument `#1` is a METAFONT pair, such as (x, y).

Transforms provided by MFPIC.

`\rotate{`θ`}` Rotates around origin by θ degrees

`\rotatearound{`p`}{`θ`}`	Rotates around point p by θ degrees
`\turn[`p`]{`θ`}`	Rotates around point p (origin is default) by θ degrees
`\mirror{`p_1`}{`p_2`}`	
`\reflectabout{`p_1`}{`p_2`}`	Reflects about the line p_1`--`p_2
`\shift{`p`}`	Shifts origin by the vector p
`\scale{`s`}`	Scales uniformly by a factor of s
`\xscale{`s`}`	Scales only the X coordinates by a factor of s
`\yscale{`s`}`	Scales only the Y coordinates by a factor of s
`\zscale{`p`}`	Scales uniformly by magnitude of p, and rotates by angle of p
`\xslant{`s`}`	Skew in X direction by the multiple s of Y
`\yslant{`s`}`	Skew in Y direction by the multiple s of X
`\zslant{`s`}`	See `zslanted` in `grafdoc.tex`
`\boost{`χ`}`	Special relativity boost by χ
`\xyswap`	Reflects in the line $Y = X$

When any of these commands is issued, the effect is to transform all subsequent figures (within the enclosing `coords` or `mfpic` environment). In particular, attention may need to be paid to whether these transformations move (part of) the figure outside the space allotted by the `\mfpic` command parameters.

A not-so-obvious point (even to the creators of MFPIC) is that if several of these transformations are applied in succession, then figures are transformed as if the transformations were applied in the reverse order, similar to the application of prefix macros (as well as application of transformations in mathematics: T_1T_2z means apply T_1 to the result, after applying T_2 to z).

9.2 *Transformation of Paths.*

In the previous section we discussed transformations of the METAFONT coordinate system. Those macros affect the *drawing* of paths and other figures, but do not change the actual paths. We will explain the distinction after introducing the following two macros

`\store{`⟨*path variable*⟩`}{`⟨*path*⟩`}`

This stores the following ⟨*path*⟩ in the specified METAFONT ⟨*path variable*⟩. Any valid METAFONT variable name will do, in particular, any sequence of letters and underscores will do. The stored path may later be used as a figure macro using `\mfobj` (below).

`\mfobj{`⟨*path expression*⟩`}`

The ⟨*path expression*⟩ is a previously stored path variable, or a valid METAFONT expression combining such variables and/or constant paths. This allows the use of path vatiables or expressions as figure macros, permitting all prefix operations, etc.. Here's some oversimplified uses of `\store` and `\mfobj`:

```
\store{f}{\circle{...}}          % store a circle
\dotted\mfobj{f}                 % now draw it dotted
\hatch\mfobj{f}                  % and hatch its interior
% Store two curves:
```

```
\store{f}{\curve{...}}
\store{g}{\curve{...}}
% Store two combinations of them:
\store{h}{\mfobj{f..g..cycle}}     % a MF path expression
\store{k}{%
  \lclosed\connect
  \mfobj{f}\mfobj{g}
  \endconnect}
\dotted\mfobj{f}                   % draw the first dotted
\dotted\mfobj{g}                   % then the second
\shade\mfobj{h}                    % now shade one combination
\hatch\mfobj{k}                    % and hatch the other
```

Getting back to coordinate transforms. If one changes the coordinate system and then stores and draws a curve, say by

```
\coords
  \rotate{45 deg}
  \store{xx}{\rect{(0,0),(1,1)}}
  \dashed\mfobj{xx}
\endcoords
```

one will get a transformed picture, but the object `\mfobj{xx}` will contain the simple, unrotated rectangular path and drawing it later (outside the `coords` environment) will prove that. This is because the `coords` environment works at the drawing level, not at the definition level. In oversimplified terms, `\dashed` invokes the transformation, not `\store`. The following transformation prefixes provide a means of actually creating and storing a transformed path.

`\rotatepath{(`$\langle x\rangle$`,`$\langle y\rangle$`),`$\langle\theta\rangle$`}`...
`\shiftpath{(`$\langle dx\rangle$`,`$\langle dy\rangle$`)}`...
`\scalepath{(`$\langle x\rangle$`,`$\langle y\rangle$`),`$\langle s\rangle$`}`...
`\xscalepath{`$\langle x\rangle$`,`$\langle s\rangle$`}`...
`\yscalepath{`$\langle y\rangle$`,`$\langle s\rangle$`}`...
`\slantpath{`$\langle y\rangle$`,`$\langle s\rangle$`}`...
`\xslantpath{`$\langle y\rangle$`,`$\langle s\rangle$`}`...
`\yslantpath{`$\langle x\rangle$`,`$\langle s\rangle$`}`...
`\reflectpath{`$\langle p_1\rangle$`,`$\langle p_2\rangle$`}`...
`\xyswappath`...

`\rotatepath` rotates the following path by $\langle\theta\rangle$ degrees about point `(`$\langle x\rangle$`,`$\langle y\rangle$`)`. After the commands:

```
\store{xx}{\rotatepath{(0,0), 45}\rect{(0,0),(1,1)}}
```

the object `\mfobj{xx}` contains an actual rotated rectangle, as drawing it will prove. The above macro, and the five that follow are extremely useful (and better than `coords` environments) if one needs to draw a figure, together with many slightly different versions of it.

`\shiftpath` shifts the following path by the horizontal amount $\langle dx \rangle$ and the vertical amount $\langle dy \rangle$.

`\scalepath` scales (magnifies or shrinks) the following path by the factor $\langle s \rangle$, in such a way that the point `(`$\langle x \rangle$`,`$\langle y \rangle$`)` is kept fixed. That is

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

is essentially the same as `\rect{(0,0),(2,2)}`, while

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

is the same as `\rect{(-1,-1),(1,1)}`. In both cases the rectangle is doubled in size. In the first case the lower left corner stays the same, while in the second case the the upper right corner stays the same.

`\xscalepath` is similar to `\scalepath`, but only the x-direction is scaled, and all points with first coordinate equal to $\langle x \rangle$ remain fixed. `\yscalepath` is similar, except the y-direction is affected.

`\slantpath` applies a slant transformation to the following path, keeping points with second coordinate equal to $\langle y \rangle$ fixed. That is, a point p on the path is moved right by an amount proportional to the height of p above the line $y = \langle y \rangle$, with s being the proportionality factor. Vertical lines in the path will acquire a slope of $1/s$, while horizontal lines stay horizontal.

`\xslantpath` is an alias for `\slantpath`

`\yslantpath` is similar to `\xslantpath`, but exchanges the roles of x and y coordinates.

`\reflectpath` returns the mirror image of the following path, where the line determined by the points $\langle p_1 \rangle$ and $\langle p_2 \rangle$ is the mirror.

`\xyswappath` returns the path with the roles of x and y exchanged. This is almost like `\reflectpath{(0,0),(1,1)}`, and produces the same result if the x and y scales of the picture are the same. However, `\reflectpath` compensates for such different scales (so the path shape remains the same), while `\xyswappath` does not (so that after a swap, verticals become horizontal and horizontals become vertical). One cannot have both.

10. Parameters.

There are many parameters in MFPIC which the user can modify to obtain different effects, such as different arrowhead size or shape. Most of these parameters have been described already in the context of macros they modify, but they are all described together here.

Many of the parameters are stored by TEX as dimensions, and so are available even if there is no METAFONT file open; changes to them are not subject to the usual TEX rules of scope however: they are only limited by the `mfpic` environment, if they are set inside one. This is for consistency: other parameters are stored by METAFONT (so the macros to change them will have no effect unless a METAFONT file is open) and the changes are subject to METAFONT's rules of scope—to the MFPIC user, this means that changes inside the `\mfpic` ... `\endmfpic` environment are local to that environment, but other TEX groupings have no effect on scope. Some commands (notably those that set the axismargins and `\tlabel` parameters) change both TEX parameters and METAFONT parameters, and it is important to keep then consistent.

`\mfpicunit`

This TEX dimension stores the basic unit length for MFPIC pictures—the x and y scales in the `\mfpic` macro are multiples of this unit. The default value is `1pt`.

`\pointsize`

This TEX dimension stores the diameter of the circle drawn by the `\point` macro and the diameter of the symbols drawn by `\plotsymbol` and by `\plot`. The default value is `2pt`.

`\pointfilltrue` and `\pointfillfalse`

This TEX boolean switch determines whether the circle drawn by `\point` will be filled (`true`) or open (outline drawn, background erased). The default is `true`. This value is local to any TEX group.

`\pen{`⟨*drawpensize*⟩`}`
`\drawpen{`⟨*drawpensize*⟩`}`
`\penwd{`⟨*drawpensize*⟩`}`

Establishes the width of the normal drawing pen. The default is `0.5pt`. This width is stored by METAFONT. The shading dots and hatching pen are unaffected by this. There exist three aliases for this command, the first two to maintain backward compatibility, the last one for consistency with other dimension changing commands.

`\shadewd{`⟨*dotdiam*⟩`}`

Sets the diameter of the dots used in the shading macro. The drawing and hatching pens are unaffected by this. The default is `0.5pt`

`\hatchwd{`⟨*hatchpensize*⟩`}`

Sets the line thickness used in the hatching macros. The drawing pen and shading dots are unaffected by this. The default is `0.5pt`.

`\polkadotwd{`⟨*polkadotdiam*⟩`}`

Sets the diameter of the dots used in the `\polkadot` macro. The default is `5pt`.

`\headlen`

This TEX dimension stores the length of the arrowhead drawn by the `\arrow` macro. The default value is `3pt`.

`\axisheadlen`

This TEX dimension stores the length of the arrowhead drawn by the `\axes`, `\xaxis` and `\yaxis` macros, and by the macros `\axis` and `\doaxes` when applied to the parameters `x` and `y`. The first name is for compatability; both reference the same TEX dimension. The default value is `5pt`.

`\sideheadlen`

This TEX dimension stores the length of the arrowhead drawn by the `\axis` and `\doaxes` macros when applied to `l`, `b`, `r` or `t`. The default value is `0pt`.

`\headshape{`⟨*hdwdr*⟩`}{`⟨*hdten*⟩`}{`⟨*hfilled*⟩`}`

Establishes the shape of the arrowhead drawn by the `\arrow` and `\axes` macros. The value of ⟨*hdwdr*⟩ is the ratio of the width of the arrowhead to its length; ⟨*hdten*⟩ is the tension of the Bézier curves; and ⟨*hfilled*⟩ is a METAFONT boolean value indicating whether the arrowheads are to be filled (if `true`) or open. The default values are 1, 1, `false`, respectively. The ⟨*hdwdr*⟩, ⟨*hdten*⟩ and ⟨*hfilled*⟩ values are stored by METAFONT. Setting ⟨*hdten*⟩ to "`infinity`" will make the sides of the arrowheads straight lines.

`\dashlen`, `\dashspace`

These TEX dimensions store, respectively, the length of dashes and the length of spaces between dashes, for lines drawn by the `\dashed` macro. The `\dashed` macro may adjust the dashes and the spaces between by as much as $\frac{1}{n}$ of their value, where n is the number of spaces appearing in the curve, in order not to have partial dashes at the ends. The default values are both `4pt`. The dashes will actually be longer (and the spaces shorter) by the thickness of the pen used when they are drawn.

`\dashlineset`, `\dotlineset`

These macros provide convenient standard settings for the `\dashlen` and `\dashspace` dimensions. The macro `\dashlineset` sets both values to `4pt`; the macro `\dotlineset` sets `\dashlen` to `1pt` and `\dashspace` to `2pt`.

`\hashlen`

This TEX dimension stores the length of the axis hash marks drawn by the `\xmarks` and `\ymarks` macros. The default value is `4pt`.

`\shadespace`

This TEX dimension establishes the spacing between dots drawn by the `\shade` macro. The default value is `1pt`.

`\darkershade`, `\lightershade`

These macros both multiply the `\shadespace` dimension by constant factors, 5/6 and 6/5 respectively, to provide convenient standard settings for several levels of shading.

`\polkadotspace`

This TEX dimension establishes the spacing between the centers of the dots used in the `\polkadot` macro. The default is `10pt`.

`\dotsize`, `\dotspace`

These TEX dimensions establishes the size and spacing between the centers of the dots used in the `\dotted` macro. The defaults are `0.5pt` and `3pt`.

`\symbolspace`

Similar to `\dotspace`, this TEX dimension established the space between symbols placed by the macro `\plot{`⟨*symbol*⟩`}`.... Its default is `5pt`.

`\hatchspace`

This TEX dimension establishes the spacing between lines drawn by the `\hatch` macro. The default value is `3pt`.

`\tlabelsep{`⟨*separation*⟩`}`

This macro establishes the separation between a label and its nominal position. It affects text written with any of the commands `\tlabel`, `\tlabels`, `\axislabels` or `\plottext`. It also sets the separation between the text and the curve defined by the commands `\tlabelrect`, `\tlabeloval` or `\tlabelellipse`. The default is `0pt`.

`\mfpdataperline`

When MFPIC is reading data from files and writing it to the output file, this macro stores the maximum number of points that will be written on a single line. Its default is defined by `\def\mfpdataperline{5 }`.

`\mfpicheight`, `\mfpicwidth`

These TEX dimensions store the height and width of the figure created by the most recently completed `mfpic` environment. This might perhaps be of interest to hackers or to aid in precise positioning of the graphics. They are meant to be read-only: the `\endmfpic` command globally sets them equal to the height and width of the picture. But MFPIC does not otherwise make any use of them.

11. For Power Users Only.

`\mfsrc{`⟨*metafont code*⟩`}`

Writes the ⟨*metafont code*⟩ directly to the METAFONT file, using a TEX `\write` command. Line breaks within ⟨*metafont code*⟩ are preserved. Almost all the MFPIC drawing macros invoke `\mfsrc`. Because of the way TEX reads and processes macro arguments, not all drawing macros preserve line breaks (nor do they all need to). However, the ones that operate on long lists of pair or numeric data (for example, `\point`, `\curve`, etc.), do preserve line breaks in that data.

Using `\mfsrc` can have some rather bizarre consequences, though, so using it is not recommended to the unwary. It is, however, currently the only way to make use of METAFONT's equation solving ability. Here's an oversimplified example:

```
\mfpic[20]{-0.5}{1.5}{0}{1.5}
\mfsrc{z1=(0,0); z2-z3=(1,2); z2+2z3=(1,-1);} % z2=(1,1), z3=(0,-1)
\arc[t]{z1,z2,z3}
\endmfpic
```

Check out the sample `forfun.tex` for more realistic examples.

`\noship`

This modifier macro turns off character shipping (by METAFONT to the TFM and GF files, or by METAPOST to appropriate EPS output file) for the duration of the innermost enclosing group (e.g., for the `mfpic` environment). This is useful if all one wishes to do in the current `mfpic` environment is to make *tiles* (see below).

`\patharr{`⟨*pv*⟩`}`...`\endpatharr`

This pair of macros, acting as an environment, accumulate all enclosing paths, in order, into a path array named ⟨*pv*⟩. Any path in the array can be accessed by means of `\mfobj`. For example, after

```
\patharr{pa}
  \rect{(0,0),(1,1)}
  \circle{(.5,.5), .5}
\endpatharr
```

Then `\mfobj{pa1}` refers to the rectangle and `\mfobj{pa2}` refers to the circle.

Note: In LaTeX, this pair of macros can be used in the form of a LaTeX-style environment called `patharr` —as in `\begin{patharr}`...`\end{patharr}`.

`\tile{`⟨*tilename*⟩`,`⟨*unit*⟩`,`⟨*wd*⟩`,`⟨*ht*⟩`,`⟨*clip*⟩`}`
...
`\endtile`

In this environment, all drawing commands contribute to a *tile*. A *tile* is a rectangular picture which may be used to fill the interior of closed paths. The units of drawing are given by ⟨*unit*⟩, which should be a dimension (like `1pt` or `2in`). The tile's horizontal dimensions are 0 to ⟨*wd*⟩·⟨*unit*⟩ and its vertical dimensions 0 to ⟨*ht*⟩·⟨*unit*⟩, so ⟨*wd*⟩ and ⟨*ht*⟩ should be pure numbers. If ⟨*clip*⟩ is `true` then all drawing is clipped to be within the tile's boundary.

By using this macro, you can design your own fill patterns (to use them, see the `\tess` macro below), but please take some care with the æsthetics!

`\tess{`⟨*tilename*⟩`}`...

Tile the interior of each closed path with a tessellation comprised of *tiles* of the type specified by ⟨*tilename*⟩. There is no default ⟨*tilename*⟩; you must make all your own tiles. Tiling an open curve is technically an error, but the METAFONT code responds by drawing the path and not doing any tiling.

Tiling large regions with complicated tiles can exceed the capacity of some versions of METAPOST. There is less of a problem with METAFONT. This is not because METAFONT

has greater capacity, but because of the natural difference between bitmaps and vector graphics (i.e., PostScript).

In METAPOST, the tiles are copied with whatever color they are given when they are defined. They can be multicolored.

`\mftitle{`⟨*title*⟩`}`

Write the string ⟨*title*⟩ to the METAFONT file, and use it as a METAFONT message. (See *The* METAFONT*book*, chapter 22: *Strings*, page 187, for two uses of this.)

`\tmtitle{`⟨*title*⟩`}`

Write the text ⟨*title*⟩ to the TEX document, and to the log file, and use it implicitly in `\mftitle`.

`\newfdim{`⟨*fdim*⟩`}`

Create a new global font dimension, named ⟨*fdim*⟩, which can be used almost like an ordinary TEX dimension. One exception is that the TEX commands `\advance`, `\multiply` and `\divide` cannot be applied directly to font dimensions (nor LATEX's `\addtolength`); however, the font dimension can be copied to a temporary TEX dimension register, which can then be manipulated and copied back (using `\setlength` in LATEX, if desired). Another exception is that changes to a font dimension are global in scope. Also beware that `\newfdim` uses font dimensions from a single font, the `dummy` font, which most TEX systems ought to have. (You'll know if yours doesn't, because MFPIC will fail upon loading!) Also, implementations of TEX differ in the number of font dimensions allowed per font. Hopefully, MFPIC won't exceed your local TEX's limit. All of MFPIC's basic dimension parameters are font dimensions. We have lied slightly when we called them "TEX dimensions".

`\setmfpicgraphic{`⟨*filename*⟩`}`

This is the command that is invoked to place the graphic created. See `mppicdoc.tex` for a discussion of its use and its default definition. It is a user-level macro so that it can be redefined in unusual cases. It operates on the output of the following macro:

`\setfilename{`⟨*file*⟩`}{`⟨*num*⟩`}`

MFPIC's figure inclusion code ultimately executes `\setmfpicgraphic` on the result of applying `\setfilename` to two arguments: the file name specified in the `\opengraphsfile` command and the number of the current picture. Normally `\setfilename` just puts them together with the "`.`" separator (because that is the way METAPOST names its output), but this can be redefined if the METAPOST output undergoes further processing or conversion to another format in which the name is changed. Any redefinition of `\setfilename` must come before `\opengraphsfile` because that command tests for the existence of the first figure. After any redefinition, `\setfilename` must be a macro with two arguments that creates the actual filename from the above two parts. It should also be completely expandable, which can be tested by issuing the command `\message{***\setfilename{filename}{1}***}`. What you should see on the terminal between the triple asterisks is only the filename, and no unexpanded TEX commands. See `mppicdoc.tex` for a possibly instructive example.

`\getmfpicoffset{`⟨*filename*⟩`}`

This command is automatically invoked after `\setmfpicgraphic` to store the offset of the lower left corner of the figure in the macros `\mfpicllx` and `\mfpiclly`. If `\setmfpicgraphic` is redefined then this may also have to be redefined. Typically, defining it to be empty (i.e., `\def\getmfpicoffset{}`) will work if either both or neither of the options mplabels and truebbox are in effect.

11.1 *For Hackers Only.*

MFPIC employs a modified version of LaTeX's `\@ifnextchar` that not only skips over spaces when seeking the next character, but also skips over `\relax` or tokens that have been `\let` equal to it. This is because, in contexts where we try to preserve lines, we make the end-of-line character active and set it equal to `\relax`. Since it is hard to predict in what context a macro will be used, this gives code like

```
\function
[s1.2]{0,2,.1}{ x**2 }
```

the same behavior in both. One consequence is that putting `\relax` to stop a command from seeing a "`[`" as the start of an optional argument will not work for MFPIC commands. The same holds for the "`*`" in those few commands that have a star-form, and also for other commands that look ahead (`\tlabel` looks for a "`(`" starting off the location, and macros that operate on lists of data look ahead for "`\datafile`"). This may all be moot, because I can't think of an MFPIC command that doesn't have mandatory argument(s) following the look-ahead location. If a "`\relax`" appeared in such a place, and it was not skipped, an error would result from reading it as the next argument.

V. Acknowledgements.

Tom would like to thank all of the people at Dartmouth as well as out in the network world for testing MFPIC and sending him back comments. He would particularly like to thank:

Geoffrey Tobin for his many suggestions, especially about cleaning up the METAFONT code, enforcing dimensions, fixing the dotted line computations, and speeding up the shading routines (through this process, Geoffrey and Tom managed to teach each other many of the subtleties of METAFONT), and for keeping track of MFPIC for nearly a year while Tom finished his thesis;

Bryan Green for his many suggestions, some of which (including his rewriting the `\tcaption` macro) ultimately led to the current version's ability to put graphs in-line or side-by-side; and

Uwe Bonnes and Jaromir Kuben, who worked out rewrites of MFPIC during Tom's working hiatus and who each contributed several valuable ideas.

Some credit also belongs to Anthony Stark, whose work on a FIG to METAFONT converter has had a serious impact on the development of many of MFPIC's capabilities.

Finally, Tom would like to thank Alan Vlach, the other TeXnician at Berry College, for helping him decide on the format of many of the macros, and for helping with testing.

Dan Luecking would like to echo Tom's thanks to all of the above, especially Geoffrey Tobin and Jaromir Kuben. And to add the names Taco Hoekwater, for comments, advice and suggestions, and Zaimi Sami Alex for suggestions and interest shown.

But mostly, he'd like to thank Tom Leathrum for starting it all.

VI. Changes History.

See the file `changes.tex` for a somewhat sporadic and rambling history of changes to MFPIC. See the file `whats.new` for a list of any known problems.

VII. Appendices

1. Summary of Options

Unless otherwise stated, any of the command forms will be local to the current `mfpic` environment if used inside. Otherwise it will affect all later environments.

Option:	Command form(s):	Restrictions:
metapost	`\usemetapost`	Command must come before `\opengraphsfile`. Incompatible with metafont option.
metafont	`\usemetafont`	The default. Command must come before `\opengraphsfile`. Incompatible with metapost option.
mplabels	`\usemplabels`, `\nomplabels`	Requires metapost, should be used with truebbox. If command is used inside an `mfpic` environment, it should come before any `\tlabel` commands.
truebbox	`\usetruebbox`, `\notruebbox`	Has no effect without metapost; should be used with mplabels. Command should not be used inside any `mfpic` environment because it is needed by the start-up code of `\mfpic`.
clip	`\clipmfpic`, `\noclipmfpic`	No restrictions.
centeredcaptions	`\usecenteredcaptions`, `\nocenteredcaptions`	If command is used inside an `mfpic` environment, it should come before `\tcaption`.
debug	`\mfpicdebugtrue`, `\mfpicdebugfalse`	To turn on debugging while `mfpic.tex` is loading, issue `\def\mfpicdebug{}`.

draft
final
nowrite
	`\mfpicdraft`
	`\mfpicfinal`
	`\mfpicnowrite`
		Should not be used together. Command forms should come before `\opengraphsfile`

2. Plotting styles for \plotdata

When \plotdata passes from one curve to the next, it increments a counter and uses that counter to select a dashpattern, color, or symbol. It uses predefined dashpattern names dashtype0 through dashtype5, or predefined color names colortype0 through colortype7, or predefined symbols pointtype0 through pointtype8. Here follows a description of each of these variables.

Under \dashedlines, we have the following dashpatterns:

Name	Pattern	Meaning
dashtype0	0pt	solid line
dashtype1	3pt,4pt	dashes
dashtype2	0pt,4pt	dots
dashtype3	0pt,4pt,3pt,4pt	dot-dash
dashtype4	0pt,4pt,3pt,4pt,0pt,4pt	dot-dash-dot
dashtype5	0pt,4pt,3pt,4pt,3pt,4pt	dot-dash-dash

Under \coloredlines, we have the following colors. Except for black and red, each color is altered as indicated. This is an attempt to make the colors more equal in visibility against a white background. (The success of this attempt varies greatly with the output or display device.)

Name	Color	(r,g,b)
colortype0	black	$(0, 0, 0)$
colortype1	red	$(1, 0, 0)$
colortype2	blue	$(.2, .2, 1)$
colortype3	orange	$(1, .66, 0)$
colortype4	green	$(0, .8, 0)$
colortype5	magenta	$(.85, 0, .85)$
colortype6	cyan	$(0, .85, .85)$
colortype7	yellow	$(.85, .85, 0)$

Under \pointedlines and \datapointsonly, the following symbols are used. Internally each is referred to by the numeric name, but they are identical to the more descriptive name. Syntactically, all are metafont path variables.

Name	Description
pointtype0	Star
pointtype1	Triangle
pointtype2	SolidCircle
pointtype3	Plus
pointtype4	Square
pointtype5	SolidDiamond
pointtype6	Cross
pointtype7	Circle
pointtype8	SolidTriangle

R

S

T