MFPIC: Pictures in TEX
with Metafont and MetaPost

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

2003/11/14

Contents

MFPIC version: 0.7 beta.

*G.Tobin@latrobe.edu.au
†luecking@uark.edu

1 Introduction

1.1 Why?

Tom got the idea for MFPIC[1] 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 graphic objects into a DVI file using TEX `\special`'s. However, this technique seemed far from TEX's ideal of device independence because different TEX output drivers recognize different `\special's`, and handle them in different ways.

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 was, of course, PICTEX, which is wonderfully flexible and general, but its most obvious feature was its speed—or rather lack of it. Processing a single picture in PICTEX could 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 the almost universal 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. This gets us away from device independence, but many users were not so much concerned with that as with having a convenient way to have text and pictures described in the same document.

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.

1.2 Who?

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.

Somewhere in the mid 1990's the development passed to Geoffrey Tobin who kept things going for several years.

The addition of METAPOST support was carried out by Dan Luecking around 1997–99. He is also responsible for all other additions and changes since then, with help from Geoffrey and a few others mentioned in the Acknowledgements.

[1] If you're wondering how to pronounce 'MFPIC': I always say 'em-eff-**pick**', speaking the first two letters. —DHL.

1.3 What?

See `README.txt` for a list and a brief explanation of each of the files. 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).

The readme file that accompanies MFPIC gives some guidence on the proper location for the installation of these files.

1.4 How?

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 (perhaps nothing!) 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. If you process your pictures with METAPOST there is nothing to do in this regard.

Here is an example of the process: for the sample file `pictures.tex`[2], 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.[3]) Typically, you just type

```
mf pics.mf
```

or, to use a particular printer mode such as `ljfour`, possibly something like

```
mf '\mode:=ljfour; input pics.mf'
```

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.) Typically, you just run

```
gftopk pics.600gf
```

(or possibly `gftopk pics.600gf pics.600pk` or `gftopk pics.600gf pics.pk`).

Now you have the font (the `.pk` file) and font metric file (the `.tfm`) 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`). You can delete `pics.600gf` and `pics.log`.

If you use MFPIC with the metapost option (this would require you to edit `pictures.tex` or `lapictures.tex`. See chapter 2 for how to do this), then `pics.mp` is produced, and you need

[2]Read `mfpguide.pdf` for examples of minimal MFPIC input files.

[3]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/`.

to replace the METAFONT/GFTOPK steps with the single step of running METAPOST. (Read your METAPOST documentation on how to do this.[4]) Typically just

```
mpost pics.mp
```

or possibly `mp pics.mp` .

After reprocessing `pictures.tex` with TEX you should then be able to run dvips on the resulting DVI file and print or view its POSTSCRIPT output. It pdfTEX is used instead of TEX on the second run, you should be able to view the resulting PDF file with the pictures included.

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.) 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 automatic `.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 unpredictable places) before you can see any later changes. It might be wise to write a shell script (batch file) that (1) runs METAFONT, (2) runs GFTOPK if step 1 returns no error, (3) deletes the `.tfm` if the `.pk` file does not exist. 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 or METAPOST steps.

There are also somewhat subtle circumstance under which you can skip the second TEX step after editing a file that has gone through the above process. Listing the exact cirumstances is rather involved, so it is recommended that you always repeat the TEX step if changes have been made.

What makes MFPIC work? When you run TEX on the file `pictures.tex`, the MFPIC macros issue TEX `\write commands`, writing METAFONT (or METAPOST) commands to a file `pics.mf` (or `pics.mp`). 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 source and the resulting METAFONT file, that MFPIC drawing macros translate almost directly into similar METAFONT/METAPOST commands, defined in one of the files `grafbase.mf` or `grafbase.mp`. The labels and captions, however, are placed on the graph by TEX using box placement techniques similar to those used in LATEX's `picture` environment (except when option mplabels is in effect, in which case METAPOST places the labels).

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

2 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. Any option not among those given below will be passed on to the GRAPHICS package, provided the metapost option has been used.

If the file `mfpic.cfg` exists, it will be input just before all options are processed. You can create such a file containing an `\ExecuteOptions` command to execute any options you would like to have as default. Actual options to `\usepackage` will override these defaults, of course. And so will any of the commands below.

If the file `mfpic.usr` exists, it will be input at the end of the loading of MFPIC. The user can create such a file containing any of the commands of this section that he would like to have as default.

2.1 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 3.1). They should not be used together in the same document. (Actually, they can but one needs to close one output file and open another. Moreover, it hasn't ever been seriously tested, and it wasn't taken into consideration in writing most of the macros.) If the command form `\usemetapost` is used in a LaTeX 2ε document, it must come in the preamble. Because of the timing of actions by the BABEL package and by older versions of `supp-pdf.tex` (input by `pdftex.def` in the GRAPHICS package), when pdfLaTeX is used MFPIC should be loaded and `\usemetapost` (if used) declared before BABEL is loaded.

2.2 mplabels, `\usemplabels`, `\nomplabels`

Causes all label creation commands to write their contents to the output file. It has no effect on the `\tcaption` command. In this case labels are handled by METAPOST and can be rotated. It requires METAPOST, and will be be ignored without it (METAFONT cannot handle labels). It may also produce an error either from TeX or METAFONT. Otherwise the commands can come anywhere and affect subsequent `\tlabel` commands. When this is in effect, the labels become part of the figure and, in the default handling, they may be clipped off or covered up by later drawing elements. But see the next section on the overlaylabels option. Labels added to a picture contribute to the bounding box even if truebbox is not 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 section 3.7. 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.).

2.3 overlaylabels, \overlaylabels, \nooverlaylabels

In the past, under mplabels all text labels created by \tlabel and its relatives were added to the picture by METAPOST *as they occurred*. This made them subject to later drawing commands: they could be covered up, erased, or clipped. With this option (or after the command \overlaylabels) text labels are saved in a separate place from the rest of a picture. When a picture is completed, the labels that were saved are added on top of it. This is the way labels always behave under the metafont option, because then TeX must add the labels and there is no possibility for special effects involving clipping or erasing (at the METAFONT level).

With the metapost option, but without mplabels it has been decided to keep the same behavior (and the same code) as under the metafont option. However, when mplabels is used, there is the possibility for special effects with text, and it has always been the behavior before this version to simply place the labels as they occurred. It turns out that placing the labels at the end is cleaner and simpler to code, so I experimented with it and rejected it as a default, but now offer it as an option. With this option, MFPIC labels have almost the same behavior with or without mplabels.

2.4 truebbox, \usetruebbox, \notruebbox

Normally METAPOST outputs an EPS file with the actual bounding box of the figure. By default, MFPIC *overrides* this and sets the bounding box to the dimensions specified by the \mfpic command that produced it. (This used to be needed for TeX is to handle \tlabel commands correctly. Now, it is just for backward compatability, and for compatability with METAFONT's behavior.) It is reasonable to let METAPOST have its way, and that is what this option does. If one of the command forms is used in an mfpic environment, it affects only that environment, otherwise it affects all subsequent figures. This option currently has no effect with METAFONT, but should cause no errors.

2.5 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. It has an often unexpected interaction with truebbox. When both are in effect, METAPOST will produce a bounding box that is the intersection of two rectangles: 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.

2.6 centeredcaptions, \usecenteredcaptions, \nocenteredcaptions

Causes multiline captions created by \tcaption to have all lines centered. This has no effect on the normal LaTeX \caption command.[5] The commands can be issued anywhere. If inside an mfpic environment they should come before the \tcaption command and affect only it, otherwise they affect all subsequent figures.

[5] 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.

2.7 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.

2.8 clearsymbols, \clearsymbols, \noclearsymbols

MFPIC has two commands, `\point` and `\plotsymbol` that place a small symbol at each of a list of points. The first can place either a small filled disk or an open disk, the choice being dictated by the setting of the boolean `\pointfilltrue` or `\pointfillfalse`. The behavior of `\point` in the case of `\pointfillfalse` is to erase the interior of the disk in addition to drawing its circumference.

The second command `\plotsymbol` can place a variety of shapes, some open, some not. Its behavior until now was always simply to draw the shape without erasing the interior. Two other commands that placed these symbols, `\plotnodes` and `\plot`, had the same behavior. With this option, two of these, `\plotsymbol` and `\plotnodes`, will erase the interior of the open symbols before drawing them. Thus `\plotsymbol{SolidCircle}` still works just like `\pointfilltrue\point`, and now with this option `\plotsymbol{Circle}` behaves the same as `\pointfillfalse\point`. The `\plot` command is unaffected by this option.

2.9 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 the existence of a nonexistent file can be very slow because the entire TeX search path will 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 (but see also section 3.1). These options can be used to override this automatic detection. All the command versions should come *before* the `\opengraphsfile` command. The `\mfpicnowrite` command *must* come before it.

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 override 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.

2.10 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.

An effort has been made (as of version 0.7) to make this universal. That is, any of the commands listed above for turning options on and off will be global when issued outside an `mfpic` environment. The debug commands are exceptions; they obey all TEX scoping rules.

We have also tried to make all other MFPIC commands for changing the various parameters follow this rule: local inside `mfpic` environment, global outside. However, as of this writing I don't claim to have caught every one.

The following are special: `\usemetapost`, `\usemetafont`, `\mfpicdraft`, `\mfpicfinal`, and `\mfpicnowrite`. Their effects are always global, partly because they should occur prior to the initialization command `\opengraphsfile` (described in section 3.1). Note that `\usemetapost` may cause a file of graphic inclusion macros to be input. If this command is issued inside a group, some definitions in that file may be lost, breaking the graphic inclusion code.

3 The Macros.

In these descriptions we will often refer to 'METAFONT' when we really mean 'METAFONT or METAPOST'. This will especially be the case whenever we need to refer to commands in the two languages which are substantially the same, but occasionally we will even talk about running 'METAFONT' when we mean running one or the other to process the figures. If we need to discriminate between the two processors, (for example when they have different behavior) we will make the difference explicit.

A similar shorthand is used when referring to TeX. It should not be taken to mean plainTeX, but rather whatever version of TeX is used to process the source file: LaTeX, pdfTeX, pdfLaTeX, etc.

Many of the commands of MFPIC have optional arguments. These are denoted just as in LaTeX, with square brackets. Thus, the command for drawing a circle can be given

```
\circle{(0,0),1}
```

having only the mandatory argument, or

```
\circle[p]{(0,0),1}
```

Whenever an optional argument is omitted, the behavior is equivalent to some choice of the optional argument. In this example, the two forms have exactly the same behavior, drawing a circle centered at $(0,0)$ with radius 1. In this case we will say that `[p]` is the *default*. Another example is `\point{(1,0)}` versus `\point[3pt]{(1,0)}`. They both place a dot at the point $(1,0)$. The second one explicitly request that it have diameter `3pt`; the first will examine the length command `\pointsize`, which the user can change, but it is initialized to `2pt`. In this case we will say the default is the value of `\pointsize`, *initially* `2pt`.

Optional arguments for MFPIC commands may consist of empty brackets (completely empty, no spaces) and the default will be used. This is useful only for commands that have two optional arguments and one only wants to change from the defaults in the second one. An optional argument should normally not contain any spaces. Even when the argument contains more than one piece of data, spaces should not separate the parts. In many cases (perhaps most) this will cause no harm, but it would be better to avoid doing it altogether.

3.1 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`). 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.

It is possible (but *has not* been seriously tested) to close one file and open another, and even to change between metapost and metafont in between. If anything goes wrong with this, contact the maintainer and it might be fixed in some later version.

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

These macros open and close the mfpic environment in which most of the rest of the macros 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. One pt is a *printer's point*, which equals 1/72.27 inches or 0.35146 millimeters.

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 be sure to reprocess the TEX file after processing the .mf or .mp file, even if no other changes are made in the figure.

\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 $\langle num\rangle + 1$, 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 \mfpic and \endmfpic, you may prefer to use \begin{mfpic} and \end{mfpic}. This is by no means required: in LATEX \begin{command} invokes \command, and \end{command} invokes \endcommand, for any environment command.

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

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 without the truebbox option, except under pdfTeX or pdfLaTeX. Those TeX programs will embed the figures right in the `.pdf` output. For normal LaTeX + DVIPS, the figures are embedded by DVIPS, which must always be repeated.

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.

3.2 Figures.

3.2.1 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). There are also color functions 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. When METAPOST output is selected, figures can be drawn in any color. Several of the above mentioned parameters are colors. MFPIC provides commands to change any of these parameters.

3.2.2 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 3.7 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 circles is the value of the predefined variable `pointcolor`, and the inside of the open circles is the value of `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. Under metapost, symbols are drawn in `pointcolor`. The ⟨*size*⟩ defaults to `\pointsize` as in `\point` above. `Asterisk` consists of six line segments while `Star` is the standard closed, ten-sided polygon. The name '`\plotsymbol`' comes from the fact that the `\plot` command, which was written first, utilizes these same symbols. The command `\symbol` was already taken (standard LATEX).

The difference between `\pointfillfalse\point`... and `\plotsymbol{Circle}`... is that the inside of the circle will not be erased in the second version (i.e., whatever else has already been drawn in that area will remain visible). This is the default (for backward compatibility), but that can be changed with the commands below.

`\clearsymbols`
`\noclearsymbols`

After the first of these two commands, subsequent `\plotsymbol` commands will draw the open symbols with their interiors erased. After the second, the default behavior (described above) will be restored. These commands have no effect on `\point`. `\plotnodes` (see subsection 3.5.1) also responds to the settings made by these commands. The `\plot` command (also in subsection 3.5.1) does not.

`\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 in the specified order.

`\rect{`⟨p_0⟩`,`⟨p_1⟩`}`

Draws the rectangle specified by the points ⟨p_0⟩ and ⟨p_1⟩, these being either pair of opposite corners of the rectangle in any order.

It is occasionally helpful to know that connected paths like those produced by `\polyline` or `\rect` have a *sense* (a direction). The sense of `\polyline` is the direction determined by the order of the points. For `\rect` the sense may be clockwise or anticlockwise depending on the corners used: it begins at the first of the two points and goes horizontally from there.

\regpolygon{⟨*num*⟩}{⟨*name*⟩}{⟨eqn_1⟩}{⟨eqn_2⟩}

This produces a regular polygon with ⟨*num*⟩ sides. The second argument, ⟨*name*⟩ is a symbolic name. It can be used to refer to the vertices later. The last two arguments should be equations that position two of the vertices or one vertex and the center. The center is refered to by ⟨*name*⟩0 and the vertices by ⟨*name*⟩1 ⟨*name*⟩2, etc., going anticlockwise around the polygon. The ⟨*name*⟩ itself (without a number) will be a METAFONT variable assigned the value of ⟨*num*⟩. For example,

```
\regpolygon{5}{Meg}{Meg0=(0,1)}{Meg1=(2,0)}
```

will produce a regular pentagon with its center at $(0,1)$ and its first vertex at $(2,0)$. One could later draw a star inside it with

```
\polygon{Meg1,Meg3,Meg5,Meg2,Meg4}
```

Moreover, Meg will equal 5. The name given becomes a METAFONT variable and care should be taken to make the name distinctive so as not to redefine some internal variable.

3.2.3 A WORD ON LIST ARGUMENTS

We have seen already four MFPIC macros that take a mandatory argument consisting of a list of coordinate pairs. There are many more, and some that take a comma-separated lists of other types of items. If the lists are long, especially if they are generated by a program, it might be more convenient if one could simply refer to an external file for the data. This is possible, and one does it the following way: instead of \lines{⟨*list*⟩}, one can write

\lines\datafile{⟨*filename*⟩}

where ⟨*filename*⟩ is the full name of the file containing the data. The required format of this file and the details of this usage can be found in subsection 3.6.3. This method is available for any command that takes a comma-separated list of data as its last argument, *with the exception of those commands that adds text to the picture*. Examples of the latter are \plottext and \axislabels (subsection 3.7.1).

3.2.4 AXES, AXIS MARKS, AND GRIDS.

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

These are retained for backward compatibility, but there are more flexible alternatives below. They 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 (section 3.4). The color of the head is the value of headcolor, the shaft is drawcolor.

Unlike other commands that produce lines or curves, these do not respond to the prefix macros of sections 3.4 and 3.5. 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 3.11) but that is pretty much the only possibility for variation.

`\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 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 3.5.3). Moreover, prefix macros may be applied to `\axis` without error (see sections 3.4 and 3.5): `\dotted\axis{x}` draws a dotted *x*-axis, but `\dotted\xaxis` produces a METAFONT error. 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 command `\axismargin` described below.

Axes on the edges are drawn so that they don't cross each other. `\doaxes{lbrt}`, for example, 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.

`\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 appendix 4.6.) 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.[6] 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 they are used then `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.

[6] 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*⟩}
\hgridlines{⟨*ysep*⟩}
\vgridlines{⟨*xsep*⟩}
\gridlines{⟨*xsep*⟩,⟨*ysep*⟩}

\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*⟩. The diameter of the dot is determined by ⟨*ptsize*⟩. The default is .5bp and is hard coded in the METAFONT macros that ultimately do the drawing. Under the metapost option, the color of the dot is pointcolor. The commands \gridpoints and \lattice are synonyms for \grid.

\hgridlines draws the horizontal and \vgridlines the vertical lines through these same points. \gridlines draws both sets of lines. The thickness of the lines is set by \penwd. Authors are recommended to either reduce the pen width or change drawcolor to a lighter color for grids. Or omit them entirely: well-designed graphs usually don't need them and almost never should both horizontals and verticals be used.

\plrgrid{⟨*rsep*⟩,⟨*anglesep*⟩}
\gridarcs{⟨*rsep*⟩}
\gridrays{⟨*anglesep*⟩}
\plrpatch{⟨*rmin*⟩,⟨*rmax*⟩,⟨*rsep*⟩,⟨*tmin*⟩,⟨*tmax*⟩,⟨*tsep*⟩}
\plrgridpoints{⟨*rsep*⟩,⟨*anglesep*⟩}

\plrgrid fills the graph with circular arcs and radial lines. \gridarcs draws only the arcs, \gridrays only the radial lines. \plrgridpoints places a dot at all the places the rays and arcs would intersect.

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.

\plrpatch 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.

If ⟨*tsep*⟩ is larger than ⟨*tmax*⟩ − ⟨*tmin*⟩, then only the boundary rays will be drawn. If ⟨*rsep*⟩ is larger than ⟨*rmax*⟩ − ⟨*rmin*⟩, then only the boundary arcs will be drawn.

The color used for rays and arcs is drawcolor, and for dots pointcolor. The advice about \gridlines holds for \plrgrid as well.

3.2.5 CIRCLES AND ELLIPSES.

\circle[⟨*format*⟩]{⟨*specification*⟩}

Draws a circle. Starting with MFPIC version 0.7, there are 4 different ways to specify a circle, so \circle can be given an optional argument that determines what data is specified in the mandatory argument.

\circle[p]{$\langle c \rangle$,$\langle r \rangle$}
\circle[c]{$\langle c \rangle$,$\langle p \rangle$}
\circle[t]{$\langle p_1 \rangle$,$\langle p_2 \rangle$,$\langle p_3 \rangle$}
\circle[s]{$\langle p_1 \rangle$,$\langle p_2 \rangle$,$\langle \theta \rangle$}

The optional arguments produce circles according to the following descriptions.

[p] The *Polar form* is the default. The data in the mandatory argument should then be the center c and radius r of the circle.

[c] The *Center-point form*. In this case the data should be the center and one point on the circumference.

[t] The *Three-point form*. The data are three points that do not lie in a straight line.

[s] The *point-sweep*. The data are two points on the circle, followed by the angle of arc between them.

These optional arguments are also used in the \arc command (see subsection 3.2.7). The \circle command draws the whole circle which the equivalent \arc command draws only part of. The sense of the circle produced is anticlockwise except in the case [t], where it is the direction determined by the order of the three points, and the case [s], where it is determined by $\langle \theta \rangle$: clockwise if negative, anticlockwise if positive.

\ellipse[$\langle \theta \rangle$]{$\langle c \rangle$,$\langle r_x \rangle$,$\langle r_y \rangle$}

Draws an ellipse with the x radius $\langle r_x \rangle$ and y radius $\langle r_y \rangle$, centered at the point $\langle c \rangle$. The optional parameter $\langle \theta \rangle$ provides a way of rotating the ellipse by $\langle \theta \rangle$ degrees anticlockwise around its center.

3.2.6 CURVES.

\curve[$\langle$*tension*$\rangle$]{$\langle p_0 \rangle$,$\langle p_1 \rangle$,...}

Draws a smooth path through the specified points, in the specified order. It is ‘smooth’ in two ways: it never changes direction abruptly (no ‘corners’ or ‘cusps’ on the curve), and it tries to make turns that are not too sharp. This latter property is acheived by specifying (to METAFONT) that the tangent to the curve at each listed point is to be parallel to the line from that point’s predecessor to its successor.

The optional $\langle$*tension*$\rangle$ influences *how* smooth the curve is. The special value infinity (in fact, usually anything greater than about 10), makes the curve not visibly different from a 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. The default value of the tension is 1 when MFPIC is loaded, but that can be changed with the following command.

\settension{$\langle$*num*$\rangle$}

This sets the default tension for all commands that take an optional tension parameter.

\cyclic[$\langle$*tension*$\rangle$]{$\langle p_0 \rangle$,$\langle p_1 \rangle$,...}

Draws a cyclic (i.e., closed) METAFONT Bézier curve through the specified points, in the specified order. It uses the same procedure as \curve, but treats the first listed point as having the last as its predecessor and the last point has the first as its successor. 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`.

`\fncurve[`⟨*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 typically 1.2 (actually 1.2 times the value set with `\settension`). For those who know something about METAFONT, this 'tension' is not the same as the METAFONT notion of tension, the tension in the `\curve` command, but it functions in a similar fashion. In this case it can be any positive number, but only values greater than or equal to 1 guarantee the property of never doubling back.

3.2.7 CIRCULAR ARCS.

`\arc[`⟨*format*⟩`]{`⟨*specification*⟩`}`

Draws a circular arc specified as determined by the ⟨*format*⟩ optional parameter. This macro and `\circle` are 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*⟩`}`
`\arc[t]{`⟨p_0⟩`,`⟨p_1⟩`,`⟨p_2⟩`}`
`\arc[p]{`⟨c⟩`,`⟨θ_1⟩`,`⟨θ_2⟩`,`⟨r⟩`}`
`\arc[a]{`⟨c⟩`,`⟨r⟩`,`⟨θ_1⟩`,`⟨θ_2⟩`}`
`\arc[c]{`⟨c⟩`,`⟨p_1⟩`,`⟨θ⟩`}`

The optional arguments produce arcs according to the following descriptions.

`[s]` The *point-Sweep form* is the default format. It draws the circular arc starting from the point ⟨p_0⟩, ending at the point ⟨p_1⟩, and covering an arc angle of ⟨*sweep*⟩ degrees, measured anticlockwise 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 anticlockwise). A negative value of ⟨*sweep*⟩ gives a clockwise arc from ⟨p_0⟩ to ⟨p_1⟩.

`[t]` The *Three-point form* draws the circular arc which passes through all three points given, in the order given. Internally, this is converted to two applications of the point-sweep form.

`[p]` The *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 anticlockwise from the positive x axis.

`[a]` The *Alternate polar form* draws the arc of a circle with center $\langle c\rangle$ and radius $\langle r\rangle$, starting at the angle $\langle\theta_1\rangle$ and ending at the angle $\langle\theta_2\rangle$. Both angles are measured anticlockwise from the positive *x* axis. This is provided because it seems a more reasonable order of arguments, and matches the order `\sector` requires (see subsection 3.2.8 below). The `p` option is retained for backward compatibility.

`[c]` The *Center-point form* draws the circular arc with center $\langle c\rangle$, starting at the point $\langle p_1\rangle$, and sweeping an angle of $\langle\theta\rangle$ around the center from that point. (This and the point sweep form are the basic methods of handling arcs—the previous three formats are translated to one of these two before drawing.)

3.2.8 OTHER FIGURES.

`\turtle{`$\langle p_0\rangle$`,`$\langle v_1\rangle$`,`$\langle v_2\rangle$`,...}`

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

`\sector{`$\langle c\rangle$`,`$\langle r\rangle$`,`$\langle\theta_1\rangle$`,`$\langle\theta_2\rangle$`}`

Draws the sector, from the angle $\langle\theta_1\rangle$ to the angle $\langle\theta_2\rangle$ inside the circle with center at the point $\langle c\rangle$ and radius $\langle r\rangle$, where both angles are measured in degrees anticlockwise 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*.[7]

`\makesector`

The `\sector` command requires the center of the arc as one of its arguments. But if one doesn't know that center (say one only knows three points the arc connects) then even though the arc can be drawn, `\sector` cannot. The `\makesector` command, when followed by any `\arc` command, will find the center and connect it to the two ends of the arc. It will actually attempt to do the same with any path that follows, but the 'center' it finds (if it finds one) will usually be meaningless.

3.2.9 BAR CHARTS AND PIE CHARTS.

`\barchart[`$\langle$*start*$\rangle$`,`$\langle$*sep*$\rangle$`,`$\langle r\rangle$`]{`$\langle$*h-or-v*$\rangle$`}{`$\langle$*list*$\rangle$`}`
`\bargraph...`
`\gantt...`
`\histogram...`
`\chartbar{`$\langle$*num*$\rangle$`}`
`\graphbar{`$\langle$*num*$\rangle$`}`
`\histobar{`$\langle$*num*$\rangle$`}`

The macro `\barchart` computes a bar chart or a Gantt chart. It does not draw the bars, but only defines their rectangular paths which the user may then draw or fill or both using the `\chartbar` macros (see below). Since bar charts have many names, `\bargraph` and `\histogram` are provided

[7] This apparently was unintended, but we now have to live with it so as not to break existing `.tex` files.

as synonyms. The macro `\gantt` is also a synonym; whether a Gantt chart or bar chart is created depends on the data.

$\langle$*h-or-v*$\rangle$ should be `v` if you want the ends of the bars to be measured vertically from the x-axis, or `h` if they should be measured horizontally from the y-axis. $\langle$*list*$\rangle$ should be a comma-separated list of numbers and/or pairs giving the coordinates of the end(s) of each bar. A number c is interpreted as the pair $(0,c)$; a pair (a,b) is interpreted as an interval giving the ends of the bar (for Gantt diagrams). The rest of this description refers to the `h` case; the `v` case is analogous.

By default the bars are 1 graph unit high (thickness), from $y = n-1$ to $y = n$. Their width and location are determined by the data. The optional parameter consists of three numeric parameters separated by commas. $\langle$*start*$\rangle$ is the y-coordinate of the bottom edge of the first bar, $\langle$*sep*$\rangle$ is the distance between the bottom edges of successive bars, and $\langle$*r*$\rangle$ is the fraction of $\langle$*sep*$\rangle$ occupied by each bar. The default behavior corresponds to `[0,1,1]`. In general, bar number n will be from $y = \langle start\rangle + (n-1) * \langle sep\rangle$ to $y = \langle start\rangle + (n-1+\langle r\rangle) * \langle sep\rangle$

Notice the bars are numbered in order from bottom to top. You can reverse them by making $\langle$*sep*$\rangle$ negative, and making $\langle$*start*$\rangle$ the top edge of the first bar.

The fraction $\langle$*r*$\rangle$ should be between -1 and 1. A negative value reverses the direction from the ‘leading edge’ of the bar to the ‘trailing edge’. For example, if one bar chart is created with

```
\barchart[1,1,-.4]{h}{..}
```

and another with

```
\barchart[1,1,.4]{h}{..}
```

both having the same number of bars, then the first will have its first bar from $y = 1$ to $y = 1 - .4 = .6$, while the second will have its first bar adjacent to that one, from 1 to $1 + .4$. Similarly the next bars will be above and below $y = 2$, etc. This makes it easy to draw bars next to one another for comparison.

The macro `\chartbar` (synonyms `\graphbar`, `\ganttbar`, and `\histobar`) takes a number from 1 to the number of elements in the $\langle$*list*$\rangle$ and draws the rectangular path. This behaves just like any other figure macro, and the prefix macros from section 3.5 may be used to give adjacent bars contrasting colors, fills, etc.

`\piechart[`$\langle$*dir*$\rangle\langle$*angle*$\rangle$`]{`$\langle$*c*$\rangle$`,`$\langle$*r*$\rangle$`}{`$\langle$*list*$\rangle$`}`
`\piewedge[`$\langle$*spec*$\rangle\langle$*trans*$\rangle$`]{`$\langle$*num*$\rangle$`}`

The macro `\piechart` also does not draw anything, but computes the `\piewedge` regions described below. The first part of the optional parameter, $\langle$*dir*$\rangle$, 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 $\langle$*angle*$\rangle$ parameter, and the change the direction to counter-clockwise by specifying `a` for $\langle$*dir*$\rangle$. 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 $\langle$*list*$\rangle$ 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 $\langle$*c*$\rangle$ for the center and a positive number $\langle$*r*$\rangle$ 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.

3.2.10 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.

3.3 Colors

3.3.1 SETTING THE DEFAULT COLORS.

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

These macros set the default color 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`. For the `\point`, `\plotsymbol` and `\grid` commands use `\pointcolor`, 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}`.

3.3.2 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 METAPOST color functions have been defined in `grafbase.mp`:

`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 numeric g (a gray level) to the corresponding 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 the same as `gray(0.5)`).

3.3.3 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.

MFPIC translates

```
\fillcolor[cmyk]{1,.3,0,.2}
```

into the equivalent of

```
\fillcolor{cmyk(1,.3,0,.2)}.
```

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.

3.3.4 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`.

3.3.5 COLOR IN METAFONT

METAFONT was never meant to understand colors, but it certainly can be taught the difference between black and white and, to a limited extent, various grays. Starting with version 0.7, MFPIC will no longer generate an error when a color-changing command is used under the metafont option. Instead, when possible, the variables that represent colors in METAPOST will be converted to a numeric value between 0 and 1 in METAFONT. When possible (for example, when a region is filled) the numeric will be interpreted as a gray level and shading (see subsection 3.5.2) will be used to approximate the gray. In other cases (drawing or dashing of curves, placing of points or symbols, filling with a pattern of hatch lines) the number will be interpreted as black or white: a value less than 1 will cause the figure to be rendered (in black), while a value equal to 1 (white) will cause pixels corresponding to the figure to be erased.

This is still somewhat experimental and depends on adhering to certain restrictions. METAFONT's syntax does not recognize a triple of numbers as any sort of data structure, but it does allow *commands* to have any number of parameters in parentheses. So colors must be specified using the color commands such as `rgb(1,1,0)` or color names such as `yellow`, and never as a bare triple. Also, as currently written, the color names defined in `dvipsnam.mp` are not defined in METAFONT. With these provisions the same MFPIC code can often produce either gray scale METAFONT pictures or METAPOST color pictures depending only on the metapost option.

The commands `\shade` and `\gfill[gray(.75)]` (see subsection 3.5.2 for their meaning) will produce a similar shade of gray, but there is a difference. The first simply adds small dots on top of whatever is already drawn. The second, however, tries to simulate the METAPOST effect, which is to cover up whatever is previously drawn. Therefore, it first zeros all affected pixels before adding the dots to simulate gray. In particular, `\gfill[white]` should have the same effect as `\gclear`.

3.4 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 3.10.2.

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).

The (already) closed paths are those that have ‘`closed`’ in their name plus: `\rect`, `\circle`, `\ellipse`, `\sector`, `\cyclic`, `\polygon`, `\plrregion`, `\chartbar`, `\piewedge`, `\tlabelrect`, `\tlabeloval`, `\tlabelellipse`, `\tlabelcircle` and `\btwnfcn` (below).

3.4.1 CLOSURE OF PATHS.

`\lclosed`...

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

`\bclosed[`⟨*tens*⟩`]`...

This macro is similar to `\lclosed`, except that it closes an open path smoothly by drawing a Bézier curve. A Bézier is METAFONT’s natural way of connecting points into a curve, and `\bclosed` is the simplest and most efficient closure next to `\lclosed`. Moreover it usually gives a reasonably aesthetic result. Sometimes, however, one might wish a tighter connection. If that is the case, use the optional argument with a value of the tension ⟨*tens*⟩ greater than 1, the default. The command `\settension` (see subsection 3.2.6) can be used to change the default.

`\sclosed[`⟨*tens*⟩`]`...

This closes the curve by mimicking the definition of the `\curve` command. That command tries to force the curve to pass through the *n*th point in a direction parallel to the line from point $(n-1)$ to point $(n+1)$. In order to close a curve in this way, the direction at the two endpoints often has to be changed, and this changes the shape of the first and last segments of the curve. Use `\bclosed` if you don’t wish this to happen. However, `\sclosed\curve` produces the same result as `\cyclic` given the same points and tension valuse. The optional tension argument is as in the `\bclosed` command.

3.4.2 REVERSAL, CONNECTION AND SUBPATHS.

`\reverse`...

Turns a path around, reversing its sense. This will affect both the direction of arrows (e.g. bi-directional arrows can be coded with `\arrow\reverse\arrow`..., where 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, adds line segments from the trailing endpoint of one 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}`.

`\partpath{`⟨*frac1*⟩`,`⟨*frac2*⟩`}`...
`\subpath{`⟨*num1*⟩`,`⟨*num2*⟩`}`...

Both produce a part of the following path. In `\partpath` the parameters ⟨*frac1*⟩ and ⟨*frac2*⟩ should be numbers between 0 and 1. The path produced travels the same course as the path that follows, but starts at the point that is ⟨*frac1*⟩ of the original length along it, and ends at the point ⟨*frac2*⟩ of its original length. If ⟨*frac1*⟩ is greater than ⟨*frac2*⟩, the sense of the path is reversed. In `\subpath`, the two numbers should be between 0 and the number of Bézier segments in the path. This is mainly for experienced METAFONTers and provides an MFPIC interface to METAFONT's '`subpath`' operation.

As an example of `\partpath`, one can put an arrowhead (see next subsection) in the middle of a path with something like the following.

```
\arrow\partpath{0,.5}\draw...
```

3.4.3 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 anticlockwise 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.

3.5 Rendering macros

3.5.1 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 subsection 3.5.3 below) 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. The default for ⟨*color*⟩ is `drawcolor`.

To save repetition, the color used for the following commands is also `drawcolor`: `\dashed`, `\dotted`, `\plot`, `\plotnodes`, and `\gendashed`,

`\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` in subsection 3.2.2. 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`, see subsection 3.2.2) 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`. If the command `\clearsymbols` has been issued then the interiors of the open symbols are erased. The effect of something like the following is rather nice:

```
\clearsymbols
\plotnodes{Circle}\draw\polyline{...}
```

This will first draw the polyline with solid lines, and then the points listed will be plotted as open circles with the portion of the lines inside the circles erased. One sees a series of open circles connected one to the next by line segments

`\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*⟩ can be any sequence of letters and underscores. 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. If one wants dashes and space with the same length, one needs to take the size desired and increase the spaces by the thickness of the drawing pen (normally `0.5pt`) and decrease the dashes by the same amount.

If `\dashpattern` is used with an odd number of entries, a space of length `0pt` is appended. This makes the last dash in one copy of the pattern abut the first dash in the next copy.

`\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 defined will cause the curve to be drawn with a solid line, and generate a METAFONT warning, but TEX will not complain. If all the dimensions in a dash pattern are 0, `\gendashed` responds by drawing a solid curve. The same is true if the pattern has only one entry.

3.5.2 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 METAFONT code responds by drawing the path and not doing any filling. These macros replace the default rendering: 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`. Under METAFONT it approximates the color with a shade of gray, clears the interior, and then fills with a pattern of black and white pixels simulating gray.

`\gclear`...

Erases everything *inside* the subsequent closed path (except text labels under some circumstances, see section 2.2 and 2.3). Under METAPOST it actually fills with the predefined color named `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 (except text labels under some circumstances, see section 2.2 and 2.3).

`\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.[8] 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. Most of the time, however, values of `\shadespace` that differ by at least 20% will produce different patterns. The actual behavior for particular values of the parameters and particular printer resolutions cannot be predicted, and we even make no guarantee it will not change from one version of MFPIC to another.

`\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

[8] 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.

color is black. So it is optimized for small dots aligned to the pixel 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`. In METAFONT, nonblack values of `fillcolor` will produce shaded dots.

`\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 less than the line thickness, the closed path is filled with ⟨*color*⟩, as if with `\gfill`. If the first optional argument appears, both parts must be present, separated by a comma. For the color argument to be present, the other optional argument must also be present. However, if one wishes only to override the default color one can use an empty first optional argument (completely empty, no spaces).

`\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*⟩.

Hatching should normally be used very sparingly, or never if alternatives are available (color, shading). Hatching at two different angles is, however, almost the only way to fill in two regions that *automatically* shows the overlapping region.

3.5.3 CHANGING THE DEFAULT RENDERING.

Rendering is the process of converting a geometric description into a drawing. In METAFONT, this means producing a bitmap (METAFONT stores these in `picture` variables), 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 redefinition is local inside an `mfpic` environment, so it can be enclosed in braces to restrict its range. Outside an `mfpic` environment it is a global redefinition.

For example, after `\setrender{\dashed\shade}` the command `\circle{(0,0),1}` produces a shaded circle with a dashed outline. Any explicit rendering prefix overrides this default.

3.5.4 EXAMPLES.

It may be instructive, for the purpose of understanding the syntax of *shape-modifier and rendering prefixes,* 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. Thus, in the first case the path is defined (by `\lines`) then closed, then the resulting closed path is shaded, then drawn; while in the second case the order is: defined, drawn, closed, shaded. In particular, what is drawn is the path not yet closed.

3.6 Functions and Plotting.

In the following macros, expressions like $f(\mathtt{x})$, $g(\mathtt{t})$ 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).

3.6.1 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. What is allowed for the function name ⟨*fcn*⟩ is more restrictive than METAFONT's rule for variable names. Roughly speaking, it should consist of letters and underscore characters only. (In particular, for those that know what this means, the name should have no suffixes.) 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. If `\fdef` occurs inside an `mfpic` environment, it is local to that environment, otherwise it is available to all subsequent `mfpic` environments.

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 commands `begingroup` and `endgroup`. The `\fdef` command automatically supplies the grouping around the definition so the user need not type them if the entire ⟨*mf-expr*⟩ is one such compound expression. METAFONT functions can call METAFONT functions, even 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.

3.6.2 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 (subsection 3.2.6) 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*⟩`]{`⟨$x_{\min}$⟩`,`⟨$x_{\max}$⟩`,`⟨Δx⟩`}{`$f(\mathtt{x})$`}`

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

```
\function{0,pi,pi/10}{sin x}
```

draws the graph of $\sin x$ between 0 and π.

`\parafcn[`⟨*spec*⟩`]{`⟨$t_{\min}$⟩`,`⟨$t_{\max}$⟩`,`⟨Δt⟩`}{`⟨*pfcn*⟩`}`

Plots the parametric path determined by ⟨*pfcn*⟩, where ⟨*pfcn*⟩ is a METAFONT function or expression of one numeric argument `t`, returning a METAFONT *pair*. Or a pair of numeric expressions $(x(\mathtt{t}), y(\mathtt{t}))$ enclosed in parentheses and separated by a comma. The default ⟨*spec*⟩ is `s`. For example

```
\parafcn{0,1,.1}{(2t, t + t*t)}
```

plots a smooth parabola from $(0,0)$ to $(2,2)$.

`\plrfcn[`$\langle spec\rangle$`]{`$\langle\theta_{min}\rangle$`,`$\langle\theta_{max}\rangle$`,`$\langle\Delta\theta\rangle$`}{`$f$`(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 $\langle\theta_{min}\rangle$ to $\langle\theta_{max}\rangle$ in steps of $\langle\Delta\theta\rangle$. Each θ value is interpreted as an angle measured in *degrees*. In the expression f(`t`), the unknown `t` stands for θ. The default $\langle spec\rangle$ is `s`. For example

```
\plrfcn{0,90,5}{sind (2t)}
```

draws one loop of a 4-petal rosette. If one needs radian measures, use something like the following.

```
\plrfcn{0,pi*radian,pi*radian/18}{sin (2t/radian)}
```

`\btwnfcn[`$\langle spec\rangle$`]{`$\langle x_{min}\rangle$`,`$\langle x_{max}\rangle$`,`$\langle\Delta x\rangle$`}{`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 $\langle x_{min}\rangle$ and $\langle x_{max}\rangle$. Unlike the previous function macros, the default $\langle spec\rangle$ is `p`—this macro is intended to be used for shading between drawn functions, a task for which smoothness is usually unnecessary. For example

```
\shade\btwnfcn{0,180,5}{0}{sind x}
```

shades the area between first crest of a sine wave and the x-axis.

Note: the effect of `\btwnfcn` could also be accomplished with

`\lclosed\connect`
`\function{`$\langle x_{min}\rangle$`,`$\langle x_{max}\rangle$`,`$\langle\Delta x\rangle$`}{`f`(x)}`
`\reverse\function{`$\langle x_{min}\rangle$`,`$\langle x_{max}\rangle$`,`$\langle\Delta x\rangle$`}{`g`(x)}`
`\endconnect`

`\plrregion[`$\langle spec\rangle$`]{`$\langle\theta_{min}\rangle$`,`$\langle\theta_{max}\rangle$`,`$\langle\Delta\theta\rangle$`}{`$f$`(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 $\langle\theta_{min}\rangle$ to $\langle\theta_{max}\rangle$ in steps of $\langle\Delta\theta\rangle$. In the expression f(`t`), the `t` stands for θ. The region is also bounded by the angles $\langle\theta_{min}\rangle$ and $\langle\theta_{max}\rangle$, i.e. by the line segments joining the origin to the endpoints of the function. The default $\langle spec\rangle$ 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. For example

```
\shade\plrregion{0,90,5}{sind (2t)}
```

shades one loop of the 4-petal rosette.

3.6.3 PLOTTING EXTERNAL DATA FILES

`\datafile[`$\langle spec\rangle$`]{`$\langle file\rangle$`}`
`\smoothdata[`$\langle tension\rangle$`]`
`\unsmoothdata`

`\datafile` defines a curve connecting the points listed in the file $\langle file\rangle$. (The context makes it clear whether this meaning of `\datafile` or that of subsection 3.2.3 is meant.) The $\langle spec\rangle$ may be `p` to produce a polygonal path, or `s` followed by a tension value (as in `\curve`) to produce a smooth path. If no $\langle spec\rangle$ is given, the default is initially `p`, but `\smoothdata` may be used to change this.

Thus, after the command `\smoothdata[`⟨*tension*⟩`]` the default ⟨*spec*⟩ is changed to `s`⟨*tension*⟩. If the tension parameter is not supplied it defaults to `1.0` (or the value set by the `\settension` command if one has been used).

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 another use, independent of the above description. We saw in subsection 3.2.3 that 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 have that list replaced with a reference to an external data file. For example, if a file `ptlist.dat` contains two or more numerical values per line separated by whitespace, then one can draw a dot at each of the points corresponding to the first pair of numbers on each line with the following.

```
\point\datafile{ptlist.dat}
```

In fact there is no essential difference between '`\datafile[p]`' and '`\polyline\datafile`', and no difference between '`\datafile[s]`' and '`\curve\datafile`'.

Here is the full list of MFPIC macros that allow this usage of `\datafile`:

- Numeric data: `\piechart`, `\barchart`, `\numericarray`, and all the axis marks commands.
- Point or vector data: `\point`, `\plotsymbol`, `\polyline`, `\polygon`, `\fcncurve`, `\curve`, `\cyclic`, `\turtle`, `\qspline`, `\closedqspline`, `\cspline`, `\closedcspline`, `\mfbezier`, `\closedmfbezier`, `\qbeziers`, `\closedqbeziers`, and `\pairarray`.

`\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(#1,#2)}` if the data are polar coordinates. The default assumptions of the `\datafile` command (i.e., space separated numbers, the first two determining each point) correspond to the setting

```
\using{#1 #2 #3}{(#1,#2)}
```

The `\using` command cannot normally be used in the replacement text of another command. Or rather, it can be so used, but then each `#` has to be doubled. If a `\using` declaration occurs in an `mfpic` environment it is local to that environment. Otherwise it affects all subsequent ones.

`\sequence`

As a special case, you can plot any number against its sequence position, with something like `\using{#1 #2}{(\sequence,#1)}`. Here, the macro `\sequence` will take on the values `1`, `2`, etc. as lines are read from the file.

`\usingpairdefault`
`\usingnumericdefault`

The command `\usingpairdefault` restores the above default for pair data. The command `\usingnumericdefault` is the equivalent of `\using{#1 #2}{#1}`.

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 internally 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 dash patterns 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 (hey, black is a color too). This has an effect only for METAPOST. The sole exception to the general rule that all curves are drawn in `drawcolor` is the `\plotdata` command after `\coloredlines` has been issued. The command `\pointedlines` causes `\plotdata` to use `\plot` commands, cycling through nine 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 dash pattern, color or symbol corresponds to each number by default. The commands below can be used to change the default dashess, colors, or symbols.

`\reconfigureplot{dashes}{`⟨pat_1⟩`, ... ,`⟨pat_n⟩`}`
`\reconfigureplot{colors}{`⟨clr_1⟩`, ... ,`⟨clr_n⟩`}`
`\reconfigureplot{symbols}{`⟨$symb_1$⟩`, ... ,`⟨$symb_n$⟩`}`

The first argument of `\reconfigureplot` is the rendering method to change: `dashes`, `colors`, or `symbols`. The second argument is a list of dash patterns, colors, or symbols. The dash patterns should be names of patterns defined through the use of `\dashpattern`. The colors can be any color names already known to METAPOST, or defined through `\mfpdefinecolor`. The symbols can be any of those listed with the `\plotsymbol` command (see subsection 3.2.2), or any known METAFONT path variable. The colors can also be METAPOST expressions of type color, and the symbols can be expressions of type path. Within a `mfpic` environment, the changes made are local to that environment. Outside, they affect all subsequent environments.

`\defaultplot{dashes}`
`\defaultplot{colors}`
`\defaultplot{symbols}`

The command `\defaultplot` restores the built-in defaults for the indicated method of rendering in `\plotdata`.

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

3.7 Labels and Captions.

3.7.1 SETTING TEXT.

If option metafont is in effect 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*⟩`){`⟨*labeltext*⟩`}`
`\tlabel[`⟨*just*⟩`]{`⟨*pair-list*⟩`}{`⟨*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. They should *not* be separated by blank lines.

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, the second syntax may be used if mplabels is in effect, 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. The default justification is `[Bl]`.

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 and uses the given point as the center of rotation. For example, `[cr]` will place the text left of the point, while `[cr180]` will rotate it around to the right side of the point (and upsidedown, of course).

There should be no spaces before, between, or after 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 or the `\cr` 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 line, 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 space reserved for the graph is expanded to make room for it. (Note: this behavior is very much different from that of 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. Normally, the label becomes part of the picture, rather than being laid on top of it, and can be covered up by any filling macros that follow, or clipped off by `\gclear` or `\gclip`. However, under the overlaylabels option (or after the command `\overlaylabels`), labels are saved and added to the picture at the very end. This may prevent some special effects, but it makes the behavior of labels much more consistent through all the 12 permissable settings of the options metapost, mplabels, clip, and truebbox.

`\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. If mplabels is not in effect at the time of this command, the rotation part will be saved in case that option is turned on later, but a warning message will be issued. Without mplabels, the rotation is ignored by `\tlabel` .

`\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` (subsection 3.7.2) and related commands.

`\axislabels{`⟨*axis*⟩`}[`⟨*just*⟩`]{{`⟨*text*$_1$⟩`}`⟨*n*$_1$⟩`,{`⟨*text*$_2$⟩`}`⟨*n*$_2$⟩`,...}`

This command places the given TEX text (⟨*text*$_k$⟩) 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 under mplabels, but they must *not* be individually enclosed in braces. (This requirement is new with version 0.7; prior to that pairs in braces didn't work reliably anyway.) 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, and `\plottext` is converted to it internally.

`\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. If it issued before `\opengraphsfile`, its contents are saved and written by that command. Because of the way METAPOST handles `verbatimtex` material, the effects cannot be constrained by any grouping unless one places TEX grouping commands within ⟨*TEX-cmds*⟩.

`\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. The separation between the bottom of the picture and the caption can be changed by increasing or decreasing the skip `\mfpiccaptionskip` (a 'rubber' length in Lamport's terminology).

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 3.11) might be a convenience for plain TeX users who want to roll their own caption macros.

3.7.2 CURVES SURROUNDING TEXT

`\tlabelrect[`⟨*rad*⟩`][`⟨*just*⟩`]`⟨*pair*⟩`{`⟨*text*⟩`}`
`\tlabelrect*...`

This and the following two methods of surounding a bit of text with a curve share some common characteristics which will be described here. The commands all take an optional argument that can modify the shape of the curve. After that come arguments exactly as for the `\tlabel` command except that only a single point is permitted, not a list. (So ⟨*pair*⟩ is either of the form `(`⟨*x*⟩`,`⟨*y*⟩`)` or the same enclosed in braces, or for mplabels a pair expression in braces.) After processing the surrounding curve, a `\tlabel` is applied to those arguments unless a `*` is present. In order for the second optional argument to be recognized as the second, the first optional argument must also be present. An empty first optional argument is permitted, causing the default value to be used. The default for the justification parameter is `cc`, for compatibility with past MFPIC versions in which these commands all centered the figure around the point and no justification parameter existed. This default can be changed with the `\tlpathjustify` command below.

The plain rectangle version produces a frame separated from the text on all sides by the amount defined with `\tlabelsep`. All other versions produce the smallest described curve that contains this rectangle.

These commands may be preceded by prefix macros (see the sections 3.4 and 3.5, above). They all have a '`*`-form' which produces the curve but omits placing the text. All have the effect of rendering the path *before* placing any text. For example, `\gclear\tlabelrect...` will clear the rectangle and then place the following text in the cleared space.

The optional argument of `\tlabelrect`, ⟨*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 rectangle mentioned above. There is one special case for the optional argument ⟨*rad*⟩: if the keyword '`roundends`' is used instead of a dimension, the radius will be chosen to make the nearest quarter circles just meet, so the narrow side of the rectangle is a half circle.

`\tlabeloval[`⟨*mult*⟩`][`⟨*just*⟩`]`⟨*pair*⟩`{`⟨*text*⟩`}`
`\tlabeloval*...`

This is similar to `\tlabelrect`, except it draws an ellipse. The ellipse is calculated to have the same ratio of width to height as the rectangle mentioned 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.

`\tlabelellipse[`⟨*ratio*⟩`][`⟨*just*⟩`]`⟨*pair*⟩`{`⟨*text*⟩`}`
`\tlabelellipse*...`
`\tlabelcircle[`⟨*just*⟩`]`⟨*pair*⟩`{`⟨*text*⟩`}`
`\tlabelcircle*...`

Draws the smallest ellipse centered at the point that encompasses the rectangle defined above, and that has a ratio of width to height equal to ⟨*ratio*⟩ then places the text. The default ratio is 1, which produces a circle. We also provide the command `\tlabelcircle`, which take only the `[`⟨*just*⟩`]` optional argument. Internally, it just processes any `*` and calls `\tlabelellipse` with parameter 1.

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 quarter-circles at the corners to be indented rather than convex. In the other cases, there is no visible effect, but in all cases the sense of the curve is reversed.

`\tlpathjustify{`⟨*just*⟩`}`

This can be used to change the default justification for `\tlabelrect` and friends. The ⟨*just*⟩ parameter is exactly as in `\tlabeljustify` in subsection 3.7.1.

3.8 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.

3.9 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 the value of the dimension `\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.

3.10 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.

3.10.1 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.

Transforms provided by MFPIC.

`\rotate{`⟨θ⟩`}`	Rotates around origin by ⟨θ⟩ degrees
`\rotatearound{`⟨*point*⟩`}{`⟨θ⟩`}`	Rotates around point ⟨*point*⟩ by ⟨θ⟩ degrees
`\turn[`⟨*point*⟩`]{`⟨θ⟩`}`	Rotates around point ⟨*point*⟩ (origin is default) by ⟨θ⟩
`\mirror{`⟨p_1⟩`}{`⟨p_2⟩`}`	Same as `\reflectabout`
`\reflectabout{`⟨p_1⟩`}{`⟨p_1⟩`}`	Reflect about the line ⟨p_1⟩`--`⟨p_2⟩
`\shift{`⟨*pair*⟩`}`	Shifts origin by the vector ⟨*pair*⟩

`\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{`⟨*pair*⟩`}`	Scales uniformly by magnitude of ⟨*pair*⟩, and rotates by angle of ⟨*pair*⟩
`\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{`⟨*pair*⟩`}`	See `zslanted` in `grafdoc.tex`
`\boost{`⟨χ⟩`}`	Special relativity boost by χ, see `boost` in `grafdoc.tex`
`\xyswap`	Exchanges the values of x and y.

An arbitrary METAFONT transformation can be implemented with

`\applyT{`⟨*transformer*⟩`}`

This is mainly for METAFONT hackers. This applies 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). Any code that satisfies METAFONT's syntax for a ⟨*transformer*⟩ (see D. E. Knuth, "The METAFONTbook") is permitted, although no effort is made to correctly write TEX special characters nor to preserve linebreaks in the code.

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 is that if several of these transformations are applied in succession, then the most recent is applied first, so that figures are transformed as if the transformations were applied in the reverse order of their occurrence. This is similar to the application of prefix macros (as well as application of transformations in mathematics: T_1T_2z usually means to apply T_1 to the result of T_2z).

3.10.2 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 two macros for storing and reusing figures.

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

This stores the following ⟨*path*⟩ in the specified METAFONT ⟨*path variable*⟩. Any valid METAFONT symbolic token will do, in particular, any sequence of letters or underscores. You should be careful to make the name distinctive to avoid overwriting the definition of some internal variable. The stored path may later be used as a figure macro using `\mfobj` (below). The ⟨*path*⟩ may be any of the figure macros (such as `\curve{(0,0),(1,0),(1,1)}`) or the result of modifying it. For example.

```
\store{pth}\lclosed\reverse\curve{(0,0),(1,0),(1,1)})
```

In fact, `\store` is a prefix macro that does nothing to the following curve except store it. It acts as a rendering macro with a null rendering, so the curve is not made visible unless other rendering macros appear before or after it. It is special in that it is the only prefix macro that allows the following path to be an argument, that is, enclosed in braces. This is solely to support past MFPIC versions in which `\store` was *not* defined as a prefix macro.

`\mfobj{`⟨*path expression*⟩`}`
`\mpobj{`⟨*path expression*⟩`}`

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

```
\store{my_f}{\circle{...}}              % Store a circle.
\dotted\mfobj{my_f}                     % Now draw it dotted,
\hatch\mfobj{my_f}                      % and hatch its interior
% Store two curves:
\store{my_f}{\curve{...}}
\store{my_g}{\curve{...}}
% Store two combinations of them:
\store{my_h}{\mfobj{my_f--my_g--cycle}} % a MF path expression
\store{my_k}{%
  \lclosed\connect                      % a combination path created from
  \mfobj{my_f}\mfobj{my_g}              %   mfpic commands.
  \endconnect}
\dotted\mfobj{my_f}                     % Draw the first dotted,
\dotted\mfobj{my_g}                     % then the second.
\shade\mfobj{my_h}                      % Now shade one combination.
\hatch\mfobj{my_k}                      % and hatch the other
```

The two forms `\mfobj` and `\mpobj` are absolutely equivalent.

It should be noted that every MFPIC figure is implicitly stored in the object `curpath`. So you can use `\mfobj{curpath}` and get the path defined by the most recent sequence of prefix macros and figure.

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, but not `\store`. More precisely, MFPIC prefix macros have an input and an output and a side effect. The input is the output of whatever follows it, the output can be the same as the input (the case for rendering prefixes) or modified version of that (the closure prefixes). The side effect is the drawing (dashing, filling) of the path, appending of an arrowhead, etc.. These side effects have to know where to place their marks, so a computation is invoked that converts the user's graph coordinates into METAFONT's drawing coordinates. The previous transformation macros work by modifying the parameters used in this computation.

The following transformation prefixes provide a means of actually creating and storing a transformed path. In the terms just discussed, their input is a path, their output is the transformed path, and they have no side effects (other than invoking the default rendering if no rendering prefix was previously provided).

`\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`...
`\transformpath{`$\langle \mathit{transformer}\rangle$`}`...

`\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 similar in some respects to `\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 when the scales are different.

For METAFONT or METAPOST power users, `\transformpath` can take any 'transformer' and transform the following path with it. Here, a *transformer* is anything that can follow a path and create a new path. Examples are `scaled`, `shifted (1,1)`, and `rotatedabout (0,1)`.

All these prefixes change only the path that follows, not any rendering of it that follows. For example:

```
\gfill\rotatepath{(0,0),90}\dashed\rect{(0,0),(1,1)}
```

will not produce a rotated dashed rectangle. Rather the original rectangle will be dashed, and the rotated rectangle will be filled.

3.11 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 local to TEX groups only if set inside an `mfpic` environment otherwise they are global. 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 or open (outline drawn, inside erased). The default is `true`: filled. This value is local to any TEX group inside an `mfpic` environment. Outside such it is global.

`\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. Publishers generally recommended authors to use at least a width of one-half point for drawings submitted for publication.

`\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`, and the value is stored by METAFONT.

`\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`, and the value is stored by METAFONT.

`\polkadotwd{`⟨*polkadotdiam*⟩`}`

Sets the diameter of the dots used in the `\polkadot` macro. The default is `5pt`, and the value is stored by METAFONT.

`\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`.

`\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. These values are all stored by METAFONT.

`\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, $5/6 = .833333$ and $6/5 = 1.2$ 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 macro `\polkadot`. 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 establishes 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`. The value is stored by both TeX and METAFONT.

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

This macro establishes a uniform offset that applies to all labels. It affects text written with any of the commands `\tlabel`, `\tlabels`, `\axislabels` or `\plottext`. The default is to have both horizontal and vertical offsets of `0pt`. The values are stored by both TEX and METAFONT.

`\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 in the output file. 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.

3.12 For Advanced Users.

3.12.1 POWER USERS.

`\qspline{`⟨*list*⟩`}`
`\closedqspline{`⟨*list*⟩`}`
`\cspline{`⟨*list*⟩`}`
`\closedcspline{`⟨*list*⟩`}`

These are alternate ways of defining curves. In each case, ⟨*list*⟩ is a comma separated list of points. These represent not the points the curve passes through, but the *control points*. The first two produce quadratic B-splines and the last two produce cubic B-splines. If you don't know what B-splines are, or don't know what control points are, it is recommended you not use these commands.

`\cbclosed`...
`\qbclosed`...

These are prefix macros for closing curves. The first closes with a cubic B-spline, the second with a quadratic B-spline. They will close any given curve, but the command `\cbclosed` is meant to close a cubic B-spline (see above). That is, `\cbclosed\cspline` should produce the same result as `\closedcspline` with the same argument. The corresponding statements are true of `\qbclosed`: it is meant to close a quadratic B-spline and `\qbclosed\qspline` should produce the same result as `\closedqspline` with the same argument.

The power user, having noticed that `\curve` and `\cyclic` insert some direction modifiers into the path created, may have decided that there is no MFPIC command to create a simple METAFONT default style path, for example `(1,1)..(0,1)..(0,0)..cycle`. If so, he or she has forgotten about `\mfobj`: the command

```
\mfobj{(1,1)..(0,1)..(0,0)..cycle}
```

will produce, in the `.mf` file, exactly this path, but surround it with the TEX wrapping needed to make MFPIC's prefix macro system work. However, the syntax of more complicated paths can be extremely lengthy, so we offer this interface:

`\mfbezier[`⟨*tens*⟩`]{`⟨*list*⟩`}`
`\closedmfbezier[`⟨*tens*⟩`]{`⟨*list*⟩`}`

This connects the points in the list with the path join operator `..tension` ⟨*tens*⟩`..`. If the tension option `[`⟨*tens*⟩`]` is omitted, the value set by `\settension` (initially 1) is used. One can get a cyclic path by prepending `\bclosed` (with matching tension option), but it will not produce the same result as `\closedmfbezier`. These are cubic Bézier's (but you know that if you are a power user). Quadratic Béziers (as in LATEX's picture environment) can be obtained with the following:

`\qbeziers[`⟨*tens*⟩`]{`⟨*list*⟩`}`
`\closedqbeziers[`⟨*tens*⟩`]{`⟨*list*⟩`}`

Note the plural, to indicate that they will draw a series of quadratic Béziers. In the ⟨*list*⟩, the first, third, fifth, etc., are the points to connect, while the second, fourth, etc., are the control points. The open version requires an ending point, and so needs an odd number of points in the list. The closed version assumes the first point is the ending, and so requires an even number in the list. The curve will not automatically be smooth. That depends on the choice of the control points.

`\mfsrc{`⟨*metafont code*⟩`}`
`\mfcmd{`⟨*metafont code*⟩`}`
`\mflist{`⟨*metafont code*⟩`}`

These all write the ⟨*metafont code*⟩ directly to the METAFONT file, using a TEX `\write` command. Line breaks within ⟨*metafont code*⟩ are preserved.[9] Almost all the MFPIC drawing macros invoke one of these. 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. The difference in these is minor: `\mfsrc` writes its argument without change, `\mfcmd` appends a semicolon ('`;`') to the code, while `\mflist` surrounds its argument with parentheses and then appends a semicolon.

Using these can have some rather bizarre consequences, though, so 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 a more realistic example.

[9] Under most circumstances, but not if the command (plus its argument) is part of another macro

`\setmfvariable{`⟨*type*⟩`}{`⟨*name*⟩`}{`⟨*value*⟩`}`
`\setmpvariable{`⟨*type*⟩`}{`⟨*name*⟩`}{`⟨*value*⟩`}`

These formerly internal MFPIC macros can be use to define symbolic names for any METAFONT or METAPOST variable type. They are interchangeable; you can use either one with or without the metapost option. As an example of their use, since dimensions are numeric data types in METAFONT, the command

```
\setmfvariable{numeric}{my_dim}{7pt}
```

would set the METAFONT variable `my_dim` to the value `7pt`. After that, `my_dim` can be used in any *drawing* command where a dimension is required:

```
\plotsymbol[my_dim]{Triangle}\rect{(0,0),(1,1)}
```

will plot the rectangle with small triangles spaced `7pt` apart.

You can define paths this way (`\setmfvariable{path}{X}{(0,0)..(1,1)..(0,1)}`), but the ⟨*value*⟩ has to be valid METAFONT path construction syntax, *not* something like `\rect{...}`. You need `\store` if you want to set a variable to an MFPIC path. However, defined either way, they can be used in `\mfobj`.

A variable defined this way is local to the `mfpic` environment it is contained in. It is in fact local to any METAFONT group. In MFPIC, only `\connect` ... `\endconnect` and `\mfpic` ... `\endmfpic` create METAFONT groups in the graph file.

`\noship`
`\stopshipping`
`\resumeshipping`

`\stopshipping` turns off character shipping (by METAFONT to the TFM and GF files, or by METAPOST to appropriate EPS output file) until `\resumeshipping` occurs. If you want just one character not shipped, just use `\noship` inside 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*⟩. A path array is a collection of paths with a common base name indexed by integers from 1 to the number of paths. 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{pa[1]}` refers to the rectangle and `\mfobj{pa[2]}` refers to the circle. In case explicit numbers are used, METAFONT allows `pa1` as an abbreviation for `pa[1]`. However, if a numeric variable or some expression is used (e.g., `pa[n+1]`) the square brackets are required.

This command can only be used in an `mfpic` environment. The definitions it makes are, however, global.

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}`.

`\pairarray{`⟨*var*⟩`}`⟨*list-of-points*⟩
`\numericarray{`⟨*var*⟩`}`⟨*list-of-numbers*⟩

These enable the simultaneous definition of pair and numeric variables. For example, after

```
\pairarray{X}{(0,1),(1,1),(0,0),(1,0)}
```

the variables `X1`, `X2`, `X3`, and `X4` are equal to the given points in that order. And then

```
\polyline{X1,X2,X3,X4}
```

will draw the lines connecting these four points. The index may optionally be put in square brackets and may be separated from the name by any number of spaces. If a numeric expression is used instead of an explicit number, square brackets *must* surround it: `X[1+1]`, `X[2]`, `X2` and `X 2` are all the same. The arrays are defined locally if these commands occur in an `mfpic` environment, global otherwise. In all arrays, the variable `X` itself (not followed by any digit or brackets) is a number equal to the number of elements in the array.

Array variables may be used only where the values are processed only by METAFONT or METAPOST, they are unknown to TeX. In particular, they cannot be used in commands that position text unless mplabels is in effect.

Several commands in MFPIC define arrays of objects that can be used in other commands. The main ones are `\piechart` and `\barchart`. These arrays are always global. Using `\piechart` causes the following arrays to become defined:

- `piewedge`, a path array describing the wedges of the chart. The command `\piewdge{`⟨*num*⟩`}` (without optional argument) is almost exactly the same as `\mfobj{piewedge[`⟨*num*⟩`]}`.
- `pieangle`, a numeric array, the starting angle of each wedge.
- `piedirection`, a pair array, the unit vectors pointing in the directions of the centers of the wedges. If `\pieangle1` is 0 and `pieangle2` is 90 degrees, then `piedirection1` is $(\cos 45, \sin 45)$.

Using `\barchart` causes the following arrays to become defined. The exact meaning depends on whether bars are horizontal or vertical. The following describes horizontal bars; interchange the roles of *x* and *y* if they are vertical:

- `barstart`, the position on the *y*-axis of the leading edge of the bars.
- `barbegin`, the *x*-coordinate of the leftmost end of the bars.
- `barend`, the *x*-coordinate of the rightmost end of the bars.
- `chartbar`, the rectangular path of the bar; `chartbar1` is the rectangle with corners at `(barbegin1,barstart1)` and `(barend1,barstart+barwd)`, where `barwd` is the width (thickness) of the bar.

- `barlength`, the same as `barend`. This is for backward compatibility; the was name chosen at a time when all the bars had one side on an axis (i.e., `barbegin[n] = 0`).

`\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! The ⟨*tilename*⟩ is globally defined by this command.

`\tess{`⟨*tilename*⟩`}`...

Tile the interior of a closed path with a tessellation comprised of copies of the *tile* 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.

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

`\cutoffafter{`⟨*obj*⟩`}`...
`\cutoffbefore{`⟨*obj*⟩`}`...
`\trimpath{`⟨dim_1⟩`,`⟨dim_2⟩`}`...
`\trimpath{`⟨dim_1⟩`}`...

These are prefix macros. The first two take an 'object' (a variable in which a path was previously stored using `\store`) and uses it to trim one end off the following path. `\cutoffbefore` cuts off the part of the path before its first intersection with the object, while `\cutoffafter` cuts off the part after the last intersection. If the path does not intersect the object, nothing is cut off. If the object and the path intersect in more than one point, as little as possible (usually[10]) is cut off. This is completely reliably only when there is only one point of intersection.

The `\trimpath` macro takes two dimensions separated by commas and trims those lengths off the initial and terminal ends of the path. If only one dimension is given, that is used at both ends. This macro is essentially equivalent to applying `\cutoffafter` and then `\cutoffbefore` where the objects are circles which have radii equal to the given dimensions and which are centered at the endpoints of the path. Consequently, if the path is shorter than either dimension, it will not intersect either circle and nothing will be trimmed. Similarly, if the result of `\cutoffafter` is shorter than the first dimension, then `\cutoffbefore` will not trim any more off. The first two macros can be

[10] METAFONT's methods for finding the 'first' point of intersection do not always find the actual first one.

used to create a curve that starts or ends right at another figure without having to find the point where the two curves intersect. The third one can be used on the result to produce a curve that stops just short of the point of intersection.

`\mftitle{`⟨*title*⟩`}`

Write the string ⟨*title*⟩ to the METAFONT file, and use it as a METAFONT message. (See *The* METAFONT*book*, chapter 22, 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`. This macro forms a local group around its argument.

Since TEX is limited to 256 dimension registers, and since dimensions are so important to typesetting and drawing, it is common to use up all 256 when drawing packages are loaded. Therefore MFPIC uses font dimensions to store dimension values. The following is the command that handles the allocation of these dimensions.

`\newfdim{`⟨*fdim*⟩`}`

This create a new global font dimension named ⟨*fdim*⟩, which is a TEX control sequence (with backslash). It 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 all 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'. We arrange for them to be local to `mfpic` environments by saving their values at the start and restoring them at the end.

`\setmfpicgraphic{`⟨*filename*⟩`}`

This is the command that is invoked to place the graphic created. See appendix 4.6.3 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 usually 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. See the appendices, subsection 4.6.3 for further dicussion.

`\preparemfpicgraphic{`⟨*filename*⟩`}`

This command is automatically invoked before `\setmfpicgraphic` to make any preparations needed. The default definition is to do nothing except when the GRAPHICS package is used. That package provides no clean way to determine the bounding box of the graphic after it is included. Since MFPIC needs this information, this command redefines an internal command of the graphics package to make the data available. If `\setmfpicgraphic` is redefined then this may also have to be redefined.

`\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.

`\ifmfpmpost`

Users wishing to write code that adjusts its behavior to the graph file processor can use this to test which option is in effect. The macro `\usemetapost` sets it true and `\usemetafont` sets it false. There are no commands `\mfpmposttrue` nor `\mfpmpostfalse`, since the user should not be changing the setting once it is set: a great deal of MFPIC internal code depends on them, and on keeping them consistent with the `\opengraphsfile` commands reading of these booleans.

`\mfpicversion`

This expands to the current MFPIC version multiplied by 100. At this writing, it produces '`70`' because the version is 0.7. It can be used to test for the current version:

```
\ifx\mfpicversion\undefined \def\mfpicversion{0}\fi
\ifnum\mfpicversion>70 ... \else ... \fi
```

`\mfpicversion` was added in version 0.7.

3.12.2 HACKERS.

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.[11] 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

[11] Actually, because of a bug in previous versions, this was not true, but it is now. I hope.

look ahead for '`\datafile`'). This is not a serious problem, because there is only one command (`\smoothdata`) that takes an optional arguments but doesn't have mandatory arguments after that. If a '`\relax`' appeared after any other MFPIC command, it would be taken as an argument and an error would result. In any case, `\empty` will stop the looking ahead if it should ever be necessary.

Most of MFPIC's commands have arguments with parts delimited by commas and parentheses. In most cases this is no problem because they are written unchanged to the `.mf` and there they are parsed just fine. Some commands' arguments, however, have to be parsed by both TEX and METAFONT. Examples are `\tlabel` (sometimes, under mplabels), and `\pointdef`. One might be tempted to use METAPOST expressions there and that works fine as long as they do not contain commas or parentheses. In such cases, they can sometimes be enclosed in braces to prevent TEX seeing these elements as delimiters, but sometimes these braces might get written to the `.mf` (or `.mp`) output and cause a METAFONT (METAPOST) error. In such cases the following work-around might be possible:

```
\def\identity#1{#1}
\pointdef{A}(\identity{angle (1,2)},3)
\rect{(0,0),\A}
```

The braces prevent TEX's argument parsing from seeing the first comma as a delimiter, but upon writing to the `.mf`, the `\identity` commands are expanded and only the contents appear in the output. (TEX parses the argument to assign meanings to `\Ax` and `\Ay`.)

4 Appendices

4.1 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 Jaromír 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 Jaromír Kuben. And to add the names Taco Hoekwater, for comments, advice and suggestions, and Zaimi Sami Alex for suggestions.

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

4.2 Changes History.

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

4.3 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. If command is used inside an `mfpic` environment, it should come before `\tlabel` commands to be affected.

overlaylabels	`\overlaylabels`, `\nooverlaylabels`	Has no effect without metapost.
truebbox	`\usetruebbox`, `\notruebbox`	Has no effect without metapost.
clip	`\clipmfpic`, `\noclipmfpic`	No restrictions.
clearsymbols	`\clearsymbols`, `\noclearsymbols`	No restrictions.
centeredcaptions	`\usecenteredcaptions`, `\nocenteredcaptions`	No restrictions.
debug	`\mfpicdebugtrue`, `\mfpicdebugfalse`	To turn on debugging while `mfpic.tex` is loading, issue `\def\mfpicdebug{true}`.

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

4.4 Plotting styles for `\plotdata`

When `\plotdata` passes from one curve to the next, it increments a counter and uses that counter to select a dash pattern, color, or symbol. It uses predefined dash pattern 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. These variables must not be used in the second argument of `\reconfigureplot`, whose purpose is to redefine these variables.

Under `\dashedlines`, we have the following dash patterns:

NAME	PATTERN	MEANING
`dashtype0`	`0bp`	solid line
`dashtype1`	`3bp,4bp`	dashes
`dashtype2`	`0bp,4bp`	dots
`dashtype3`	`0bp,4bp,3bp,4bp`	dot-dash
`dashtype4`	`0bp,4bp,3bp,4bp,0bp,4bp`	dot-dash-dot
`dashtype5`	`0bp,4bp,3bp,4bp,3bp,4bp`	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	$(.66,.34,0)$
`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. (The order changed between versions 0.6 and 0.7.)

NAME	DESCRIPTION
`pointtype0`	`Circle`
`pointtype1`	`Cross`
`pointtype2`	`SolidDiamond`
`pointtype3`	`Square`
`pointtype4`	`Plus`
`pointtype5`	`Triangle`
`pointtype6`	`SolidCircle`
`pointtype7`	`Star`
`pointtype8`	`SolidTriangle`

4.5 Special considerations when using METAFONT

The most important restriction in METAFONT is on the size of a picture. Coordinates in METAFONT ultimately refer to pixel units in the font that is output. These are required to be less than 4096, so an absolute limit on the size of a picture is whatever length a row of 4096 pixels is. In fonts prepared for a LaserJet4 (600 DPI), this means about 6.8 inches. For a 1200 DPI pronter, the limit is about 3.4 inches.

A similar limit holds for numbers input, and the values of variables: METAFONT will return an error for `sin 4096`. Intermediate values can be greater (`sin (2*2048)` will cause no error), but final, stored results are subject to the limit. An MFPIC example that generated an error recently was:

```
\mfpicunit 1mm
\mfpic[10]{-3}{7}{-3.5}{5}
  \function{-4.5,4,.1}{x*x}
\endmfpic
```

The problem was the value of $(-4.5)^2 = 20.25$ in pixel units (after multiplying by the `\mfpic` scaling factor the `\mfpicunit` in inches and the DPI value): $20.25 \times 10 \times 0.03937 \times 600 > 4783$. The error did not occur at the point of creating the font, but merely at the point of storing the path in an internal variable for manipulation and drawing.

4.6 Special considerations when using METAPOST

4.6.1 REQUIRED SUPPORT

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

Under plainTEX	The file epsf.tex
Under LATEX209	The file epsf.tex or epsf.sty
Under LATEX 2ε	The package GRAPHICS or GRAPHICX
Under pdfLATEX	The package GRAPHICS or GRAPHICX with option pdftex
Under plain pdfTEX	The files supp-pdf.tex and supp-mis.tex
In all cases	The files grafbase.mp and dvipsnam.mp plus, of course, mfpic.tex (and mfpic.sty for LATEX)

The files grafbase.mp and dvipsnam.mp should be in a directory searched by METAPOST. The remaining files should be in directories searched by the appropriate TEX variant. If METAPOST cannot find the file grafbase.mp, then by default it will try to input grafbase.mf, which is generally futile (or fatal).

In case pdfLATEX is used, the graphics package is given the pdftex option. This option requires the file pdftex.def which currently inputs the files supp-pdf.tex and supp-mis.tex. The file pdftex.def is supplied with the GRAPHICS package. The other two are usually supplied with a pdfTEX distribution, and are definitely part of the ConTEXt distribution. Older versions had some bugs in connection with the BABEL package. One workaround was to load the GRAPHICS package and MFPIC before BABEL.

If the user loads one of the above required files or packages before the MFPIC macros are loaded then MFPIC will not reload them. If they have not been input, MFPIC will load whichever one it decides is required. In the LATEX 2ε case, MFPIC will load the GRAPHICS package. If the user wishes GRAPHICX, then that package must be loaded before MFPIC.

4.6.2 METAPOST IS NOT METAFONT

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

Pictures in METAPOST are stored as (possibly nested) sequences of objects, where objects are things like points, paths, contours, other pictures, etc. In METAFONT, pictures are stored as a grid of pixels. Pictures that are relatively simple in one program might be very complex in the other and even exceed memory allocated for their storage. Two examples are the \polkadot and \hatch commands. When the polkadot space and size are both too small, a \polkadot-ed region has been known to exceed METAPOST capacity, while being well within METAFONT capacity. In METAPOST the memory consumed by \hatch goes up in direct proportion to the linear dimensions of the figure being hatched, while in METAFONT it goes up in proportion to the area, and then the reverse can happen, with METAFONT's capacity exeeded far sooner that METAPOST's.

In 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.

4.6.3 GRAPHIC INCLUSION

It may be impossible to completely cater to all possible methods of graphic inclusions with automatic tests. The macro that is invoked to include the POSTSCRIPT graphic is `\setmfpicgraphic` and the user may (carefully!) redefine this to suit special circumstances. Actually, MFPIC runs the following sequence:

`\preparemfpicgraphic{`⟨*filename*⟩`}`
`\setmfpicgraphic{`⟨*filename*⟩`}`
`\getmfpicoffset{`⟨*filename*⟩`}`

The following are the default definitions for `\setmfpicgraphic`:

In plain TEX:	`\def\setmfpicgraphic#1{\epsfbox{#1}}`
In LATEX209:	`\def\setmfpicgraphic#1{\epsfbox{#1}}`
In LATEX 2ε:	`\def\setmfpicgraphic#1{\includegraphics{#1}}`
In pdfLATEX:	`\def\setmfpicgraphic#1{\includegraphics{#1}}`
In pdfTEX:	`\def\setmfpicgraphic#1{\convertMPtoPDF{#1}{1}{1}}`

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

`\DeclareGraphicsRule{.1}{eps}{.1}{}` in LATEX 2ε.

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

After running the command `\setmfpicgraphic`, MFPIC runs `\getmfpicoffset` to store the lower left corner of the bounding box of the figure in two macros `\mfpicllx` and `\mfpiclly`. All the above versions of `\setmfpicgraphic` (except `\includegraphics`) make this information available; the definition of `\getmfpicoffset` merely copies it into these two macros. What MFPIC does in the case of `\includegraphics` is to modify (locally) the definition of an internal command of the graphics package so that it copies the information to those macros, and then `\getmfpicoffset` does nothing. This internal modification is accomplished by the macro `\preparemfpicgraphic`. Changes to `\setmfpicgraphic` might require changing either or both of `\preparemfpicgraphic` and `\getmfpicoffset`. All three of these commands are fed the graphic's file name as the only argument, although only `\setmfpicgraphic` currently does anything with it.

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

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

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

As MFPIC is currently written, `\setfilename` must be *completely expandable*, which means it should contain no definitions, no assignments such as `\setcounter`, and no calculations. To test whether a proposed definition is completely expandable, put

```
\message{***\setfilename{file}{1}***}
```

after the definition in a `.tex` file and view the result on the terminal or in the `.log` file. You should see only your expected filename between the asterisks.

4.7 MFPIC and the rest of the world

4.7.1 THE LITERATURE

There are at least two places where MFPIC has garnered more than a cursory mention. The most up-to-date is a section in *The LATEX Graphics Companion* by Michel Goossens, Sebastian Rahtz and Frank Mittelbach. It describes a version prior to the introduction of METAPOST support, but it correctly describes a subset of its current commands and abilities.

The other is *TEX Unbound* by Alan Hoenig, which contains a chapter on MFPIC. Unfortunately, it describes a version that was replaced in 1996 with version 0.2.10.9. The following summarizes the differences between the description[12] found in Chapter 15 and MFPIC versions 0.2.10.9 through the current one:

`\wedge` is now renamed `\sector` to avoid conflict with the TEX command of the same name. The syntax is slightly different from that given for `\wedge`:

`\sector{(`⟨*x*⟩`,`⟨*y*⟩`),` ⟨*radius*⟩`,` ⟨*angle1*⟩`,` ⟨*angle2*⟩`}`

[12] While I'm at it: *TEX Unbound* occasionally refers to MFPIC using a logo-like formatting in which the 'MF' is in a special font and the 'I' is lowered. This 'logo' may suggest a relationship between MFPIC and PICTEX. There is no such relationship, and there is no official logo-like designation for MFPIC.

The macro `\plr{(`$\langle r_0\rangle$`,`$\langle\theta_0\rangle$`),(`$\langle r_1\rangle$`,`$\langle\theta_1\rangle$`),...}` is used to convert polar coordinate pairs to rectangular coordinates, so the commands `\plrcurve`, `\plrcyclic`, `\plrlines` and `\plrpoint` were dropped from MFPIC. Now use

`\curve{\plr{(`$\langle r_0\rangle$`,`$\langle\theta_0\rangle$`),(`$\langle r_1\rangle$`,`$\langle\theta_1\rangle$`),...}}`

instead of

`\plrcurve{(`$\langle r_0\rangle$`,`$\langle\theta_0\rangle$`),(`$\langle r_1\rangle$`,`$\langle\theta_1\rangle$`),...}`

and similarly for `\cyclic`, `\lines` and `\point` with respect to `\plrcyclic`, `\plrlines` and `\plrpoint`.

`\fill` is now renamed `\gfill` to avoid conflict with the LaTeX command of the same name.

`\rotate`, which rotates a following figure about a point, is now renamed `\rotatepath` to avoid confusion with a similar name for a transformation (see below).

`\white` is now renamed `\gclear` because `\white` is too likely to be chosen for, or confused with, a color command.

The following affine transform commands were changed from a third person indicative form (which could be confused with a plural noun) to an imperative form:

Old name:	New name:
`\boosts`	`\boost`
`\reflectsabout`	`\reflectabout`
`\rotatesaround`	`\rotatearound`
`\rotates`	`\rotate`
`\scales`	`\scale`
`\shifts`	`\shift`
`\xscales`	`\xscale`
`\xslants`	`\xslant`
`\xyswaps`	`\xyswap`
`\yscales`	`\yscale`
`\yslants`	`\yslant`
`\zscales`	`\zscale`
`\zslants`	`\zslant`

`\caption` and `\label` are now renamed `\tcaption` and `\tlabel` to avoid conflict with the LaTeX commands.

`\mfcmd` was renamed `\mfsrc` for clarity, and (in version 0.7) a new `\mfcmd` was defined, which is pretty much the same except it appends a semicolon to its argument.

There is a misprint: `\axisheadlin` should be `\axisheadlen`.

Finally, in the LaTeX template on page 496: MFPIC now supports the `\usepackage` method of loading.

4.7.2 OTHER PROGRAMS

There exists a program, FIG2MFPIC that produces MFPIC code as output. The code produced (as of this writing) is somewhat old and mostly incompatible with the description in this manual. Fortunately, it is accompanied by the appropriate versions of files `mfpic.tex` and `grafbase.mf`. Unfortunately, the names conflict with the current filenames and so they should only be used in circumstances where no substitution will occur, say in a local directory with the other sources for the document being produced. Moreover, the documentation in this manual may not apply to the code produced. However the information in *TEX Unbound* may apply.

There exist a package, CIRCUIT_MACROS, that can produce a variety of output formats, one of which is MFPIC code. One writes a file (don't ask me what it consists of) and apparently processes it with M4 and then DPIC to produce the output. The MFPIC code produced appears to be compatible with the current MFPIC.

4.8 Index of commands, options and parameters by page

4.9 List of commands by type

4.9.1 FIGURES

\arc
\axis
\btwnfcn
\chartbar
\circle
\closedcspline
\closedmfbezier
\closedqbeziers
\closedqspline
\cspline
\curve
\cyclic
\datafile
\ellipse
\fcncurve
\function
\graphbar
\histobar
\lines
\mfbezier
\mfobj
\parafcn
\piewedge
\plrfcn
\plrregion
\polygon
\polyline
\qbeziers
\qspline
\rect
\regpolygon
\sector
\tlabelcircle
\tlabelellipse
\tlabeloval
\tlabelrect
\turtle

4.9.2 FIGURE MODIFIERS

\arrow
\bclosed
\cbclosed
\connect, \endconnect
\cutoffafter
\cutoffbefore
\lclosed
\makesector
\partpath
\qbclosed
\reflectpath
\reverse
\rotatepath
\scalepath
\sclosed
\shiftpath
\slantpath
\subpath
\transformpath
\trimpath
\xscalepath
\xslantpath
\xyswappath
\yscalepath
\yslantpath

4.9.3 FIGURE RENDERERS

\dashed
\dotted
\draw
\gclear
\gclip
\gendashed
\gfill
\hatch
\lhatch
\plot
\plotdata (sort of)
\plotnodes
\polkadot
\rhatch
\shade
\tess
\thatch
\xhatch

4.9.4 LENGTHS

\axisheadlen
\dashlen
\dotsize
\dotspace
\hashlen
\hatchspace
\headlen
\mfpiccaptionskip
\mfpicheight
\mfpicunit
\mfpicwidth
\pointsize
\polkadotspace
\shadespace
\sideheadlen
\symbolspace

4.9.5 COORDINATE TRANSFORMATION

\applyT
\boost
\coords, \endcoords
\mirror
\reflectabout
\rotate
\rotatearound
\scale
\shift
\turn
\xscale
\xslant
\xyswap
\yscale
\yslant
\zscale
\zslant

4.9.6 AXES, GRIDS, AND MARKS

\axes
\axis
\axismarks
\bmarks
\doaxes
\grid
\gridarcs
\gridlines
\gridpoints
\gridrays
\hgridlines
\lattice
\lmarks
\plotsymbol
\plrgrid
\plrgridpoints
\plrpatch
\point
\rmarks
\tmarks
\vgridlines
\xaxis
\xmarks
\yaxis
\ymarks

4.9.7 SETTING OPTIONS

\clearsymbols
\clipmfpic
\mfpicdebugfalse
\mfpicdebugtrue
\mfpicdraft
\mfpicfinal
\mfpicnowrite
\nocenteredcaptions
\noclearsymbols
\noclipmfpic
\nomplabels
\nooverlaylabels
\noship
\notruebbox
\overlaylabels
\resumeshipping
\stopshipping
\usecenteredcaptions
\usemetafont
\usemetapost
\usemplabels
\usetruebbox

4.9.8 CHANGING VALUES

\axismargin

\darkershade
\dashlineset
\dashpattern
\dotlineset
\drawpen
\hatchwd
\headshape
\lightershade
\mfpicnumber
\mfplinestyle
\mfplinetype
\pen
\penwd
\polkadotwd
\setallaxismargins
\setallbordermarks
\setaxismargins
\setaxismarks
\setbordermarks
\setmfvariable
\setmpvariable
\settension
\setxmarks
\setymarks
\shadewd

4.9.9 CHANGING COLORS

\backgroundcolor
\drawcolor
\fillcolor
\hatchcolor
\headcolor
\mfpdefinecolor
\pointcolor
\tlabelcolor

4.9.10 DEFINING ARRAYS

\barchart
\bargraph
\histogram
\numericarray
\pairarray
\patharr, \endpatharr
\piechart

4.9.11 CHANGING BEHAVIOR

\clearsymbols
\coloredlines
\dashedlines
\datapointsonly
\defaultplot
\everytlabel
\makepercentcomment
\makepercentother
\mfpdatacomment
\mfpdataperline
\mfpverbtex
\noclearsymbols
\pointedlines
\pointfillfalse
\pointfilltrue
\reconfigureplot
\setrender
\smoothdata
\tlabeljustify
\tlabeloffset
\tlabelsep
\tlpathjustify
\unsmoothdata
\using
\usingnumericdefault
\usingpairdefault

4.9.12 FILES AND ENVIRONMENTS

\closegraphsfile
\mfpframe, \endmfpframe
\mfpic, \endmfpic
\opengraphsfile

4.9.13 TEXT

\axislabels
\plottext
\tcaption
\tlabels
\tlabel

4.9.14 MISC

\fdef
\getmfpicoffset
\ifmfpmpost

\mfcmd
\mflist
\mfpframed
\mfpicversion
\mfpverbtex
\mfsrc
\mftitle
\newfdim
\newsavepic
\plr
\pointdef
\preparemfpicgraphic
\savepic
\setmfpicgraphic
\setfilename
\sequence
\store
\tile, \endtile
\tmtitle
\usepic