MFPIC: Pictures in TEX
with Metafont and MetaPost[*]

Daniel H. Luecking[†]   Thomas E. Leathrum   Geoffrey Tobin

2012/12/03

Contents

MFPIC version: 1.10.

[*]Copywrite 2002–2012, Daniel H. Luecking

[†]`luecking@uark.edu`: Communications regarding MFPIC should be sent to this author. Any first-person references in this manual refer to Dr. Luecking.

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 was wonderfully flexible and general, but its most obvious feature was its speed—or rather lack of it. Processing a single picture in PiCTeX (in those days) 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 was 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 did not get along well with PK fonts, but was quite happy with METAPOST pictures. So METAPOST support was added to MFPIC. This got us a little bit away from device independence, but many users were not much concerned with that: they just wanted a convenient way to have text and pictures described in the same document file.

With the extra capabilities of POSTSCRIPT (e.g., color) and the corresponding abilities of META-POST, 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?

The original MFPIC (and still the core of the current version) 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.

1.3 What?

See the README file for a list of files in the distribution and a brief explanation of each. Only three are actually needed for full access to MFPIC's capabilities: `mfpic.dtx`, `mfpic.ins` and `grafbase.dtx`. Running LaTeX on `mfpic.ins` creates the only required files:

    `mfpic.tex` and `mfpic.sty`, the latter required only for LaTeX.
    `grafbase.mf`, required only if METAFONT will be processing figures.

---

[1] 'MFPIC' is pronounced by spelling the first two letters: 'em-eff-pick'.

grafbase.mp, dvipsnam.mp and mfpicdef.tex, needed only if METAPOST will be the processor.

The README file also gives some guidance on the proper location for the installation of these files.

1.4 How?

Some guidance on writing files that contain MFPIC figures can be found in the accompanying file mfpguide.pdf. If you use MFPIC to produce METAPOST figures the process is straightforward: run TeX (or LaTeX), then METAPOST, then TeX again. If there are no errors, then DVIPS or other DVI-to-PS converter can be run to produce viewable/printable output. You can also run DVIPDFM(X) to obtain PDF output, or even use pdfTeX instead of TeX (or pdfLaTeX instead of LaTeX) to get PDF output directly.

Here is an example of the process: for the sample file pictures.tex, first run TeX on it (or run LaTeX on lapictures.tex). You may see a message from MFPIC that there is no file pics.1, but TeX will continue processing the file anyway. When TeX is finished, you will now have a file called pics.mp. This is the METAPOST file containing the descriptions of the pictures for pictures.tex. You need to run METAPOST on pics.mp (Read your METAPOST manual to see how to do this.[2]) Typically, you just type

    mpost pics.mp

This usually produces files named[3] pics.1, pics.2, etc., the number of files depending on the version of pictures.tex. You then reprocess pictures.tex with TeX to produce a DVI file. This file can then be processed with DVIPS (for example) to produce POSTSCRIPT output which can be printed or viewed. One can also process the DVI with DVIPDFM(X) to produce a PDF file.

If pdfTeX is used instead of TeX on the second run, you should be able to view the resulting PDF file immediately, without any further processing.

If instead you use MFPIC to produce METAFONT figures, things are a little less straightforward. The process is TeX, then METAFONT, then GFTOPK, then TeX again. After this, TeX's DVI output ought to be viewable and printable by most DVI viewers or printer drivers. For a few TeX systems there may be some prior setup needed. One needs to convince 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 drivers look in the current directory for .pk files. There may also be some setup needed to ensure that the .pk files are created at a resolution that matches that of your printer and of your DVI viewer. See the discussion in mfpguide.pdf.

If you want to test this process on the supplied sample files, edit pictures.tex removing the \usemetapost command (or edit lapictures.tex, removing the metapost option). After that, run TeX on pictures.tex (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. 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.[4]) Typically, you just type

    mf pics.mf

---

[2]The document *Some experiences on running Metafont and MetaPost*, by Peter Wilson, can be useful for beginners. Fetch CTAN/info/metafp.pdf. 'CTAN' means the Comprehensive TeX Archive Network. You can find the mirror nearest you by pointing your browser at http://www.ctan.org/ .

[3]Recent METAPOST allows one to change the default names of the output files. Current MFPIC provides an interface to that capability: see \setfilenametemplate on page 77.

[4]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.

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 that 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`).

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`, because 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 far-away places) before you can see any later changes. It might be wise to write a shell script (batch file) that runs both METAFONT and GFTOPK. It should also do some error checking and delete the `.tfm` if the `.pk` file is not produced. 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 or META-POST, 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 figure if the file has already gone through the above process. Delineating the exact cirumstances is rather involved, so it is recommended that you always repeat the TEX step if you have made changes that affect any figure.

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 `.mf` or `.mp` 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 the labels are written to the `.mp` file and handled by METAPOST).

Note: In this manual, when describing MFPIC operations, 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 program `mf` or `mpost` 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 "plain TEX", but rather whatever version of TEX is used to process the source file: plain TEX, LATEX, pdfTEX, or pdfLATEX. Also $\mathcal{AMS}$-TEX, EPLAIN and some other variants. When last tried, MFPIC didn't work with ConTEXt.

2 Options.

There are several options to the MFPIC package. These options can be turned on with certain provided commands, but under LaTeX they can also be used in the standard LaTeX \usepackage optional argument. 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 in the \usepackage command 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.

   Finally, if a file named mfpic.usr can be found, 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, plus any other TeX code.

2.1 metapost, metafont, \usemetapost, \usemetafont.

The option metapost or the command \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 (for backward compatibility of files written before METAPOST existed). Either command must come before the \opengraphsfile command (see section 4.1). 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 2_ε 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 effects only labels on the figure, not a caption added by the \tcaption command (see section 4.7.1). 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). Using this option without the metapost option may also produce an error message either from TeX or METAFONT. The command forms can be placed anywhere. If used outside an mfpic environment, they affect all subsequent \tlabel commands; inside an mfpic environment they affect all \tlabel commands in that figure.

   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 4.7. 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, the mpost command may take an option for this purpose, or an environmental variable named TEX may be set equal 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 version 0.7 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.

The commands may be used anywhere. Outside a figure they affect all subsequent figures, inside a figure they affect all subsequent text in that figure. The commands and option are ignored under the metafont option.

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.

This option is almost mandatory if you wish to use DVIPDFM(X) to convert TEX's DVI output to PDF. Both DVIPDFM and DVIPDFMX have a tendency to clip METAPOST figures to the stated bounding box. Thus, anything running outside those bounds is lost.

2.5 **clip, \clipmfpic, \noclipmfpic.**

The clip option 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 clipping rectangle (i.e., the one specified in the \mfpic command). It is possible for the actual figure to be much smaller than this bounding box (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.**

The centeredcaptions option 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. They should not be used in the argument of a \tcaption command.

---

[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 **raggedcaptions, \useraggedcaptions, \noraggedcaptions.**

The raggedcaptions option causes multiline captions created by \tcaption to have all lines ragged-right. If centeredcaptions is on, both sides will be ragged. This option can be turned off with the command \noraggedcaptions. This is the default: to have all lines except the last justified. The last line is either centered or flush left according to whether centeredcaptions is on or off.

   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. They should not be used in the argument of a \tcaption command.

2.8 **debug, \mfpicdebugtrue, \mfpicdebugfalse.**

The debug option 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. Any definition will work because MFPIC only checks whether it is defined.

2.9 **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 before version 0.7 was to always 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.

2.10 **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 4.1). 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.

2.11 **mfpreadlog, \mfpreadlog.**

From version 0.8, there exists a scheme to allow METAFONT or METAPOST to pass information back to the .tex file. This is done by writing code to the figure file requesting METAFONT to place that information in the .log file it produces. This option instructs MFPIC to read through that log file line-by-line looking for such information. Since such log files can be potentially quite lengthy, this is made an option. If the command form \mfpreadlog is used, it must come before the \opengraphsfile command, since that is when the file will be examined. At the present time, the only MFPIC facility that requires this two-way communication is \assignmfvalue (see subsection 4.12.8). If this is used, the filename given to \opengraphsfile should not be the same as the TEX source file in which this occurs, as then the wrong .log may be read.

2.12 **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. If this is ever untrue, and I don't document that fact, please let me know.

The following are special:

    \usemetapost, \usemetafont, \mfpicdraft, \mfpicfinal, \mfpicnowrite,
    and \mfpreadlog.

Their effects are always global, partly because they should occur prior to the initialization command \opengraphsfile (described in section 4.1). 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 METAFONT **and** METAPOST **Data Types.**

Since the arguments of most MFPIC drawing commands are sent to METAFONT to be interpreted, it's useful to know something about METAFONT concepts. In this chapter we will discuss some of the data types METAFONT supports. Even the casual user should know how coordinates and colors are treated and so should at least skim the next two sections. The last section can be read when the user wants to manipulate more complex objects.

METAFONT permits several different data types, and we will mainly be concerned with six of these: numeric, pair, color (METAPOST only), path, picture and boolean.[6] In METAPOST version 1.000, a tenth data type was added, cmykcolor, and the color data type can be referred to as 'rgbcolor' if one wants to emphasize the distinction.

A *variable* is a symbolic name, which can be a single letter such as A, or a descriptive name like origin. Any sequence of letters and underscores is permitted as a variable name. Numeric indexes are also allowed, provided all variables that differ only in the index have the same type. Thus A1, A2, etc., might be variables which are all of type pair. Quite a lot more is permited for variable names, but the rules are rather complex and easy to violate. MFPIC has commands for creating both simple variables and indexed variables (called *arrays*) but the casual user can get quite a lot of use out of MFPIC without ever creating or using a METAFONT variable.

METAFONT also has something akin to functions. For example, sin(1.57) might represent a function named sin receiving the parameter 1.57 as input and returning the appropriate value. Functions can take any number of parameters and return any of the data types that METAFONT supports.[7]

3.1 Numerics and pairs.

METAFONT has numeric quantities. These include lengths, such as the radius of a circle, as well as dimension units such as in (inches) and pt (points). In fact it understands all the same units that TEX does. These numeric quantities can be constants (explicit numbers) or variables (symbolic names). In fact, in and pt are symbolic names for numeric quantities.

METAFONT also has pair objects, which may be constants or variables. Constants of type pair have the form (x,y) where x and y are numbers, for example (0,0). Pairs are two-dimensional quantities used for representing either points or vectors in a rectangular (Cartesian) coordinate system.

In this manual we often represent each pair by a brief name, such as $\langle p \rangle$ or $\langle v \rangle$, the meanings of which are usually obvious in the context of the macro. These are intended to be replaced in actual use by either a pair constant or variable. The succinctness of this notation helps us to think geometrically rather than only of coordinates.

3.2 Colors.

METAPOST has the same concepts as METAFONT, but also has color objects, which may also be constants or variables. In recent MP, colors come in two flavors: rgbcolor and cmykcolor. Constants of type rgbcolor 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 (the 'rgb' model). Constants of type cmykcolor have the form (c,m,y,k) where c, m, y and k are numbers between 0 and 1 determining the relative proportions of cyan, magenta, yellow and black in the color (the 'cmyk' model).

---

[6] For the curious, there are a total of eight types (nine or ten for METAPOST). The other three are string, transform and pen. METAFONT also permits expressions that produce nothing, which is sometimes called the vacuous type, but doesn't allow (or need) variables of this type.

[7] Including the vacuous type.

A color variable is a name, like red, blue (both predefined rgb colors in METAPOST) or magenta (predefined by MFPIC to be an rgb color if METAPOST has version < 1.000, a cmyk color if the version is at least 1.000).

3.3 Paths, pictures and booleans.

Most of the things that MFPIC is designed to draw are paths. Examples of paths are circles, rectangles, other polygons, graphs of functions and splines. Because we tend to want to draw these (or fill them, or render them in other ways) we call the MFPIC commands that produce them *figure macros*. Although they are much more complex than numerics, pairs, or colors, they can still be stored in symbolic names.

Normally in MFPIC we want to create a picture, usually by rendering one or more paths. It is possible in METAFONT to store a picture in a symbolic name without actually drawing it. However, because of their complexity, objects of type picture require somewhat more care than paths or other data types. Do not expect to use stored pictures in the same way as stored paths. In fact, one should use picture variables only in those command that are explicitly designed for them. In MFPIC to date these are only \tile...\endtile and \mfpimage to store pictures, and \putmfpimage to draw copies of one. There is also \tess, but it is used only to fill a region with copies of a picture created by \tile.

The boolean data type is one of the values true or false. Variables of type boolean are symbolic names that can take either of these two values. Usually these are used to influence the behavior of some command by setting a relevant boolean variable to one or the other value.

4   The Macros.

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 "[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 requests 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".

If an MFPIC command that takes an optional argument finds only empty brackets (completely empty, no spaces), then it will use the default value. This is useful for commands that have two optional arguments and one wants the default value in the first one and some nondefault value in the second. 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 some cases this will cause no harm, but it would be better to avoid doing it altogether, because there are cases where it will cause wrong results or error messages.

4.1   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 may 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.

There may be limitations on what can be used as a filename. As of MFPIC version 1.00, we have tried to permit \jobname as part of ⟨*file*⟩. Thus we permit TeX macros, but they should expand to non-special characters. Permitting macros makes it essentially impossible for the filename to contain the backslash and brace characters. Also spaces are problematic. However other special TeX characters (for example: tilde, underscore and percent) can be used, though that is not recommended.

`\mfpic[`⟨*xfactor*⟩`][`⟨*yfactor*⟩`]{`⟨*xmin*⟩`}{`⟨*xmax*⟩`}{`⟨*ymin*⟩`}{`⟨*ymax*⟩`}`

  ...

`\endmfpic`

These macros open and close the `mfpic` environment[8] in which the drawing macros make sense. While many MFPIC commands can be used inside or outside this environment, those that actually produce visible output are required to be inside. The `\mfpic` macro also sets up the local coordinate system for the picture. The ⟨*xfactor*⟩ and ⟨*yfactor*⟩ 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 and each coordinate system unit is 1 `\mfpicunit`. If only one is specified, then they are assumed to be equal. Note that some drawing commands require equal scales to work as expected: if you try to draw a circle with different scales you will get an ellipse.

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 $(\langle xmax\rangle - \langle xmin\rangle) \cdot \langle xfactor\rangle$ times `\mfpicunit`, its height $(\langle ymax\rangle - \langle ymin\rangle) \cdot \langle yfactor\rangle$ times `\mfpicunit`, and its depth zero.

Most of MFPIC's drawing macros accept parameters which are *coordinate pairs*. A coordinate pair is a pair of numbers (x, y) enclosed in parentheses, with $\langle xmin\rangle \leq x \leq \langle xmax\rangle$ and $\langle ymin\rangle \leq y \leq \langle ymax\rangle$.[9] We will call these *graph coordinates* and refer to the numbers *x* and *y* as being *in graph units*. Things like the thickness of lines and the lengths of arrowheads are required to be expressed in actual lengths such as `1pt` or `3mm`. These will be referred to as *absolute* units.

One can scale all pictures uniformly by changing `\mfpicunit`, and scale an individual picture by changing ⟨*xfactor*⟩ and ⟨*yfactor*⟩. 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 parameters that are defined in absolute units. If you wish, you can set these to multiples of `\mfpicunit`, but it is difficult (and almost certainly unwise) to get the thickness of lines (for example) 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.

The value of these 6 parameters to `\mfpic` are available within the environment as macros: `\xfactor`, `\yfactor`, `\xmin`, `\xmax`, `\ymin` and `\ymax`.

`\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 part of the name of the graphic file that is output. The above command tells MFPIC to ignore this sequence and number the next `mfpic` figure with ⟨*num*⟩ (and the one after that ⟨*num*⟩ + 1, etc.). It is up to the user to make sure no number is repeated, as no checking is done. Numbers greater than 255 may cause errors, as TEX assumes that characters are represented by numbers no larger than that. If the first figure is to be numbered something other than 1, then, under the metapost option,

---

[8]We use the term 'environment' loosely. However, in LATEX one may use an actual `mfpic` environment. See page 13.

[9]These inequalities can be violated, usually causing something to be drawn outside the designated borders of the figure.

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.

\everymfpic{⟨*commands*⟩}
\everyendmfpic{⟨*commands*⟩}

These commands store the ⟨*commands*⟩. The first arranges for these commands to be issued at the start of every mfpic environment and the second arranges for its commands to be issued at the end of every such environment. These could be any commands that make sense inside that environment. The purpose of these commands is to save typing if there is identical setup being performed in every picture.

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

In LATEX you may prefer to use \begin{mfpic} and \end{mfpic} (instead of \mfpic and \endmfpic). This is by no means required. The sample file lapictures.tex provided with MFPIC illustrates this use of an mfpic environment in LATEX.

One should be careful using TEX groups inside mfpic environments. These can be useful to limit the scope of declarations or of changes to some variables. However, they do not limit the scope of changes to the figure file that is being written, so there is a danger that TEX and METAFONT will have different values. There are also some MFPIC commands that need to be at the outermost level. Thus, grouping should generally be avoided except for those groups provided by MFPIC commands.

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

4.2 Common objects.

The MFPIC macros that draw things can be roughly divided into two classes.

1. Those that simply cause something to be drawn. Examples of these are the \point command, which places a dot at a list of coordinates, and \gridlines, which draw coordinate lines with specified separation.
2. Those that both *define* and draw a *path*. The macros \circle, \rect, and \polyline are examples of these.

Macros of type 2 are referred to hereafter as *figure macros*, for lack of a better term. With them one can use *prefix macros* to modify various aspects of the path and how it is drawn. For example,

    \polyline{(1,2),(3,4)}

draws a line from $(1,2)$ to $(3,4)$, but

    \dotted\polyline{(1,2),(3,4)}

produces a dotted version, and

    \arrow\polyline{(1,2),(3,4)}

draws it with an arrowhead at the tip. This is not possible with \gridlines, for example. As MFPIC and the accompanying METAFONT package GRAFBASE are currently written, prefix macros can only be applied to single paths, and \gridlines produces a whole set of lines. In this manual, as each macro is introduced, if it is a figure macro, this will be explicitly stated.

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. MFPIC provides commands to change any of these parameters. When METAPOST output is selected, figures can be drawn in any color and several of the above mentioned parameters are colors. For example, drawcolor is the name of the default color used to draw curves, headcolor is used when drawing arrowheads, etc. To save repetition: all special colors for figures are initialized to black except background, which is white.

4.2.1 POINTS, LINES, AND RECTANGLES

\point[⟨*size*⟩]{⟨p_0⟩,⟨p_1⟩,...}

Draws small disks centered at the points specified in the list of ordered pairs. The optional argument ⟨*size*⟩ is an absolute dimension that determines the diameter of the disks. The default is the TEX dimension \pointsize, initially 2pt. The disks have a filled interior if the command \pointfilltrue has been issued (the initial behavior). After the command \pointfillfalse, \point commands will produce outlined circles with the interiors erased. The color of the circles is the value of the predefined variable pointcolor, and the color inside of the open circles is the value of the variable background.[10]

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

Draws small symbols centered at the points ⟨p_0⟩, ⟨p_1⟩, and so on. The symbols must be given by name, and the available symbols are:

> Asterisk, Circle, Diamond, Square, Triangle, Star, SolidCircle, SolidDiamond, SolidSquare, SolidTriangle, SolidStar, Cross, and Plus.

The names should be self-explanatory, the 'Solid' ones are filled in, the others are outlines. 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 five-pointed star formed from ten straight line segments. Cross is a × shape. The name '\plotsymbol' comes from the fact that the \plot command (see subsection 4.5.1), which was written first, utilizes these same symbols. The command \symbol was already taken (standard LATEX).

While one would rarely want to use them for this purpose, the following symbols are also available:

> Arrowhead, Crossbar, Leftbar, Rightbar, Lefthook, Righthook, Leftharpoon, Rightharpoon.

These are mainly intended for making arrows. See subsection 4.4.3 for a further description.

The difference between \pointfillfalse\point and \plotsymbol{Circle} is that the inside of the circle will not be erased in the second version, so 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 4.5.1) also responds to the settings made by these commands. The \plot command (also in subsection 4.5.1) does not.

You can design your own 'symbols'. See the discussion of arrowheads in subsection 4.4.3, and of storing paths in subsection 4.10.2.

\pointdef{⟨*name*⟩}(⟨*xcoord*⟩,⟨*ycoord*⟩)

Defines a symbolic name for an ordered pair and the coordinates it contains. ⟨*name*⟩ is any legal TEX command name *without* the backslash; ⟨*xcoord*⟩ and ⟨*ycoord*⟩ are any numbers. For example,

---

[10]METAPOST cannot actually erase. The illusion of erasing is created by painting over with background.

after the command \pointdef{A}(1,3), \A expands to (1,3), while \Ax and \Ay expand to 1 and 3, respectively. If mplabels is in effect one can use \A to specify where to place a text label, but if TEX is placing labels one must use (\Ax,\Ay). In most other cases, one can use \A where a pair or point is required.

\polyline{$\langle p_0\rangle$,$\langle p_1\rangle$,...}
\lines{$\langle p_0\rangle$,$\langle p_1\rangle$,...}

The figure macro \polyline produces connected line segments from $\langle p_0\rangle$ to $\langle p_1\rangle$, and from there to $\langle p_2\rangle$, etc. The result is an open polygonal path through the specified points, in the specified order. The macro \lines is an alias for \polyline.

\polygon{$\langle p_0\rangle$,$\langle p_1\rangle$,...}
\closedpolyline{$\langle p_0\rangle$,$\langle p_1\rangle$,...}

The figure macro \polygon produces a closed polygon with vertices at the specified points in the specified order. It works exactly like \polyline except the last point in the list is also joined to the first. The macro \closedpolyline is an alias for \polygon.

\rect{$\langle p_0\rangle$,$\langle p_1\rangle$}

This figure macro produces the closed rectangle with horizontal and vertical sides, having the points $\langle p_0\rangle$ and $\langle p_1\rangle$ as diagonally opposite corners. The same rectangle can be specified in four different ways: either pair of opposite corners in either order.

It is occasionally helpful to know that connected paths like those produced by \polyline or \rect have a *start* and a *finish* as well as *sense* (or direction). The path produced by \polyline starts at the first listed point and ends at last, having the direction determined by the order of the points. For \rect the sense may be clockwise or anticlockwise depending on the corners used: it starts by moving horizontally from the first listed point. Several MFPIC macros (such as those that add arrowheads) treat the beginning and the end of a path differently, or adjust their behavior according to the sense of the curve.

\regpolygon{$\langle num\rangle$}{$\langle name\rangle$}{$\langle eqn_1\rangle$}{$\langle eqn_2\rangle$}

This figure macro produces a closed regular polygon with $\langle num\rangle$ sides. The second argument, $\langle name\rangle$ 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 referred to by $\langle name\rangle$0 and the vertices by $\langle name\rangle$1 $\langle name\rangle$2, etc., going anticlockwise around the polygon. The $\langle name\rangle$ itself (without a number suffixed) will be a METAFONT variable assigned the value of $\langle num\rangle$. For example,

    \regpolygon{5}{Kay}{Kay0=(0,1)}{Kay1=(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{Kay1,Kay3,Kay5,Kay2,Kay4}

Moreover, Kay 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.

4.2.2   A WORD ABOUT LIST ARGUMENTS

We have seen already four MFPIC macros that take a mandatory argument consisting of an arbitrary number of coordinate pairs, separated by commas. There are many more, and some that take a comma-separated list of items of other types. 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 \polyline{⟨*list*⟩}, one can write

> \polyline\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 4.6.3. This method is available for any command that takes a comma-separated list of data (of arbitrary length) as its last argument, *with the exception of those commands that add text to the picture*. Examples of the latter are \plottext and \axislabels (subsection 4.7.1).

4.2.3   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 4.4). 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 prefix macros. They always draw a solid line (with an arrowhead unless \axisheadlen is 0pt). They *do* respond to changes in the pen thickness (see \penwd in section 4.11) 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 prefix macros. The \arrow prefix previously mentioned is an exception: these macros add an arrowhead automatically. For example, the sequence \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.

\axisline{⟨*one-axis*⟩}
\border

These are figure macros that draw the line or lines that an \axis command would draw. An \axis command is almost the equivalent of

> \arrow[l⟨*hlen*⟩]\axisline{⟨*one-axis*⟩}.

The `\axisline` command is provided as a figure macro for maximum flexibility. For example, one can use the star-form of the `\arrow` command if desired or decorate it with ones own choice of arrowhead (see subsection 4.4.3).

Also a figure macro, `\border` produces the rectangle which, if drawn, is visibly the same as the four border `\axisline`s (without heads). It is a closed path and could easily be drawn with a `\rect` command, but the `\border` command automatically adjusts for the margins set by the commands below.

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 `\doaxes`, 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{⟨one-axis⟩}{⟨num⟩}`
`\setaxismargins{⟨num⟩}{⟨num⟩}{⟨num⟩}{⟨num⟩}`
`\setallaxismargins{⟨num⟩}`

The parameter ⟨*one-axis*⟩ is one of the letters `l`, `b`, `r`, or `t`, and `\axismargin` causes the given axis to be shifted *inward* by the ⟨*num*⟩ specified (in *graph* units). 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. That is, axes can be draw in separate commands with changes to the declared margins in between. Be aware that various other commands are affected by the margin values. Examples are the already mentioned `\border`, as well as `\grid` and `\gridlines` (page 18 in this subsection).

`\xmarks[⟨len⟩]{⟨numberlist⟩}`
`\ymarks[⟨len⟩]{⟨numberlist⟩}`
`\lmarks[⟨len⟩]{⟨numberlist⟩}`
`\bmarks[⟨len⟩]{⟨numberlist⟩}`
`\rmarks[⟨len⟩]{⟨numberlist⟩}`
`\tmarks[⟨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 5.6.) The command `\axismarks{x}` is equivalent to `\xmarks` and so on for each of the six axes. (I would have used the shorter name `\marks`, but that name was already taken by eTEX.)

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`. Spaces are not needed unless a variable name is used in place of one of the numbers (see subsection 4.12.4). The number of spaces is not significant.[11] Users of this syntax should be aware that if any of the numbers is not an integer then, because of natural round-off effects, the last value might be overshot and a mark not printed there. For example, to ensure that a mark is printed at the point 1.0 on the *x*-axis, the second line below is better than the first.

```
    \xmarks{0 step .2 until 1.0}
    \xmarks{0 step .2 until 1.1}
```

\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, `ontop` for the right side axis). In these cases, no error message is produced. These words are actually METAFONT numeric variables and the variables `ontop` and `onleft`, for example, have the same value. Thus, using `ontop` for the right axis will have the same effect as `onleft`. Similarly, `onright` and `onbottom` are the same. The parameters `inside` and `outside` usually 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.

\grid[⟨*size*⟩]{⟨*xsep*⟩,⟨*ysep*⟩}
\gridpoints[⟨*size*⟩]{⟨*xsep*⟩,⟨*ysep*⟩}
\lattice[⟨*size*⟩]{⟨*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 ⟨*size*⟩. The default is the value of \griddotsize, initially `0.5pt`. 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 grid

---

[11]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}. See subsection 4.12.7 for more on for-loops in MFPIC.

lines. Or omit them entirely: well-designed graphs usually don't need them and almost never should both horizontals and verticals be used.

The above commands draw their dots and lines within the margins set by the axis margin commands on page 17.

```
\plrgrid{⟨rsep⟩,⟨anglesep⟩}
\gridarcs{⟨rsep⟩}
\gridrays{⟨anglesep⟩}
\plrpatch{⟨rmin⟩,⟨rmax⟩,⟨rsep⟩,⟨tmin⟩,⟨tmax⟩,⟨tsep⟩}
\plrgridpoints[⟨size⟩]{⟨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 (diameter ⟨size⟩) at all the places the rays and arcs would intersect. It takes an optional argument for the size of the dots, the default being \griddotsize, the same as the \grid command.

The arcs lie on circles centered at $(0,0)$ and the rays would all meet at $(0,0)$ if extended. The corresponding METAFONT commands actually draw just enough to cover the graph area and then clip them to the graph boundaries. If you don't want them clipped, use \plrpatch. Unlike the rectangular coordinate grid commands, these do not respect the axis margins (rectangular margins don't really belong with polar coordinates).

\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 color and use of \gridlines holds for \plrgrid and its relatives as well.

```
\vectorfield[⟨hlen⟩]{⟨xsp⟩,⟨ysp⟩}{⟨formula⟩}{⟨restriction⟩}
\plrvectorfield[⟨hlen⟩]{⟨rsp⟩,⟨tsp⟩}{⟨formula⟩}{⟨restriction⟩}
```

These commande draw a field of vectors (arrows). The optional argument is the length of the arrowhead, the default being the dimension \headlen, initially 3pt.

For \vectorfield, an arrow is drawn starting from each point (x,y) where x is an integer multiple of ⟨xsp⟩ and y is an integer multiple of ⟨ysp⟩. The vector field is given by ⟨formula⟩, which should be a pair-valued expression in the literal variables x and y. Typically that would be a pair of numeric expressions enclosed in parentheses and separated by a comma. The last argument is a boolean expression in the literal variables x and y, used to restrict the domain. That is, if the expression is false for some (x,y), no arrow is drawn at that point. If you do not wish to restrict the domain, type true for the restriction.

For \plrvectorfield, an arrow is drawn starting from each point with polar coordinates (r,θ) if r is an integer multiple of ⟨rsp⟩ and θ is an integer multiple of ⟨tsp⟩. In this case, the ⟨formula⟩ must be a pair-valued expression in the literal variables r and t. This should be (or produce) a pair of x and y coorinates, not a polar coordinate pair. If you have formulas $R(r,\theta)$ for the length of each vector and $T(r,\theta)$ for the angle, then the following will convert to (x,y) pairs:

```
{polar (R(r,t),T(r,t))}
```

The last argument is as in \vectorfield, except it should depend on the literal variables r and t.

In either case, the arrow is not drawn if the starting point would lie outside the borders set with \axismargins and its relatives.

The following draws a rotational field, omitting the inside of the circle of radius 1, where the arrows would be excessively long, and especially avoiding $(0,0)$ where the vector field is undefined.

```
\vectorfield[2.5pt]{.25,.25}{.5*(-y,x)/(x**2+y**2)}{x**2+y**2 >= 1}
```

The following is the same field, represented by arrows whose locations are regularly spaced in polar coordinates.

```
\plrvectorfield[2.5pt]{.25,20}{polar(.5/r,t+90)}{r >= 1}
```

4.2.4 CIRCLES, ARCS AND ELLIPSES

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

This figure macro produces a circle. Starting with MFPIC version 0.7, there are more than one way to specify a circle. In version 0.8 and later there are six ways, and one selects which one by giving \circle an optional argument that signals what data will be specified in the mandatory argument.

\circle[p]{⟨c⟩,⟨r⟩}
\circle[c]{⟨c⟩,⟨p⟩}
\circle[t]{⟨p_1⟩,⟨p_2⟩,⟨p_3⟩}
\circle[s]{⟨p_1⟩,⟨p_2⟩,⟨θ⟩}
\circle[r]{⟨p_1⟩,⟨p_2⟩,⟨r⟩}
\circle[q]{⟨p_1⟩,⟨p_2⟩,⟨r⟩}

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. A negative radius is a mathematical error, but it is accepted. It produces the same circle, with the same sense, but the starting point (normally ⟨r⟩ units to the right of the center) is ⟨r⟩ units *left* of the center.

[c] The *center-point form*. In this case the data should be the center and one point on the circumference. The circle starts at the point and has an anticlockwise sense.

[t] The *three-point form*. The data are three points that do not lie in a straight line. The circle starts at the first point and has the sense determined by the order of the points.

[s] The *point-sweep form*. The data are two points on the circle, followed by the angle of arc between them. This circle starts at the first point and has a sense determined by the angle: anticlockwise for positive angles, clockwise for negative.

[r] The *point-radius form*. The data are two points on the circle, followed by the radius. There are two circles with this data. The one that makes the angle from the first to the second point positive and less than 180 degrees is produced. The sense of the circle is normally anticlockwise starting at the first point. Using a negative radius is a mathematical error, but this command just produces the other circle with the opposite sense.

[q] The *alternative point-radius form*. The data are the same as for the [r] case, except the other circle is produced. That is, a circle starting at the first point, proceeding anticlockwise through an angle greater than 180 degrees to the second point, then along the shorter arc to the first point. Again, a negative radius produces the other circle with clockwise sense.

These optional arguments are also used in the \arc command (see below). The \circle command draws the whole circle of which the corresponding \arc command draws only a part.

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

This figure macro produces a circular arc specified as determined by the ⟨*format*⟩ optional parameter. As with \circle, 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. The starting point of each arc is at the first specified angle or point and the ending point is at the last one.

The star-form produces the complementary arc. That is, instead of the arc described below, it produces the rest of the circle from the ending point to the starting point of the arc described.

\arc[s]{⟨p_0⟩,⟨p_1⟩,⟨θ⟩}
\arc[p]{⟨c⟩,⟨θ_1⟩,⟨θ_2⟩,⟨r⟩}
\arc[a]{⟨c⟩,⟨r⟩,⟨θ_1⟩,⟨θ_2⟩}
\arc[c]{⟨c⟩,⟨p_1⟩,⟨θ⟩}
\arc[t]{⟨p_0⟩,⟨p_1⟩,⟨p_2⟩}
\arc[r]{⟨p_0⟩,⟨p_1⟩,⟨r⟩}
\arc[q]{⟨p_0⟩,⟨p_1⟩,⟨r⟩}

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 ⟨θ⟩ 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 ⟨θ⟩ 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 ⟨θ⟩ gives a clockwise arc from ⟨p_0⟩ to ⟨p_1⟩.

[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. If the first angle is less than the second, the arc has an anticlockwise sense, otherwise clockwise. A negative radius is a mathematical error, but the result is the arc on the opposite side of the circle, as if both angles were increased by 180 degrees

[a] The alternative polar form differs from the polar form above only in the order of the arguments. This seems (to me) a more reasonable order, and matches the order \sector requires (see below). The [p] option is retained for backward compatibility.

[c] The *center-point-angle form* draws the circular arc with center ⟨c⟩, starting at the point ⟨p_1⟩, and sweeping an angle of ⟨θ⟩ around the center from that point. This is the fundamental method for drawing arcs. All other methods are converted to this or the point-sweep method. Even the point sweep form is converted to this one for angles greater than 90 degrees.

[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.

[r] The *point-radius form* draws an arc on the circle that \circle[r] would produce. The arc starts at the point ⟨p_0⟩ and ends at ⟨p_1⟩. Of the two possible arcs on that circle, it produces the shorter one: the one with an angle θ less than 180 degrees measured anticlockwise around the center of the circle. A negative radius is a mathematical error, but the result is the short arc on the other circle with a clockwise sense.

[q] The *alternative point-radius form* is the same as [r] except it produces the longer arc: the one with angle θ larger than 180 degrees measured anticlockwise around the center of the circle. A negative radius is a mathematical error, but the result is the longer arc on the other circle with a clockwise sense.

For both options `[r]` and `[q]` the angle is computed and then the point-sweep method is used. If the absolute value of the radius is less than half the distance between the points, then no such arc exists. In this case, the angle is just set equal to ± 180 degrees (as if the radius were changed to half the distance).

`\sector{⟨c⟩,⟨r⟩,⟨θ₁⟩,⟨θ₂⟩}`

This figure macro produces the sector of the circle with center at the point $\langle c \rangle$ and radius $\langle r \rangle$, from the angle $\langle \theta_1 \rangle$ to the angle $\langle \theta_2 \rangle$. 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*.[12]

`\ellipse[⟨θ⟩]{⟨c⟩,⟨rₓ⟩,⟨r_y⟩}`

This figure macro produces 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. Ellipses may also be created by differentially scaling a circle and perhaps rotating the result. See subsection 4.10.2.

`\fullellipse{⟨C⟩,⟨M₁⟩,⟨M₂⟩}`
`\halfellipse{⟨M₁⟩,⟨M₂⟩,⟨M₃⟩}`
`\quarterellipse{⟨M₁⟩,⟨A⟩,⟨M₂⟩}`

For any parallelogram there is a unique ellipse incribed in it which is tangent to the sides at their midpoints. The above allows one to obtain that ellipse and parts of it. The input to `\fullellipse` is the center $\langle C \rangle$ of that parallelogram and the midpoints $\langle M_1 \rangle$ and $\langle M_2 \rangle$ of two adjacent sides. For `\halfellipse`, one supplies the midpoints $\langle M_1 \rangle$, $\langle M_2 \rangle$, and $\langle M_3 \rangle$ of three successive sides. Lastly, `\quarterellipse` requires the midpoints of two adjacents sides and the corner $\langle A \rangle$ between them. Internally, a quarter-circle is transformed to produce the quarter-ellipse and the other two are built up out of two or four such quarter-ellipses.

The reasoning behind my choice of parameters is based on the anticipated use of these commands. For example, I wanted `\quarterellipse` to be used to round a corner represented by the three points M_1, A and M_2. In order for the quarter-ellipse to have the same direction at M_1 and M_2 as the polygon M_1AM_2, the associated parallelogram has to have midpoints at M_1 and M_2.

`\plr{(⟨r₀⟩,⟨θ₀⟩),(⟨r₁⟩,⟨θ₁⟩), ...}`

When dealing with arcs and circles, it is useful to work in polar coordinates. The macro `\plr` causes METAFONT to replace 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. It must be cautioned that this wholesale conversion of a list applies only to commands that take a list consisting of an arbitrary number of points, such as `\polyline`.

The effect of `\plr` is to apply a METAFONT command, `polar`, to each point in the list, producing a new list. This METAFONT command can also be used separately in any situation where a single METAFONT point is required. For example, to connect the point $(2,3)$ to the point with polar coordinates $(1,135)$ write

    \polyline{(2,3),polar(1,135)}

This last circle-producing macro I wrote for my own use. It produces a circle associated with the hyperbolic geometry of a disk or a half-plane.

---

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

`\pshcircle{⟨center⟩,⟨radius⟩}`
`\pshcircle*{⟨center⟩,⟨radius⟩}`

This produces the circle whose hyperbolic center is at ⟨*center*⟩ and whose pseudohyperbolic radius is ⟨*radius*⟩. This all takes place inside the circle with center $(0,0)$ and radius 1 (the *unit circle*). The ⟨*center*⟩ is required to be inside the unit circle and the ⟨*radius*⟩ is required to be less than 1.

The star-form is for the *upper half-plane*, which is the set of points with positive *y*-coordinate In this case, the ⟨*center*⟩ must be in the upper half-plane and the ⟨*radius*⟩ must still be less than 1. If you are not versed in hyperbolic geometry, be warned that the actual diameter of the resulting circle is on the order of $2y/(1-R)$, where R is the ⟨*radius*⟩ and *y* is the *y*-coordinate of ⟨*center*⟩. This can be quite large even for modest values of R and *y*.

Finally, an arc-producing macro. This is also related to the hyperbolic geometry of a disk or a half-plane. The hyperbolic geometry includes a notion of distance that allow one to determine 'shortest' path between two points. This shortest path is called a 'geodesic' and it turns out (in the case of the upper half-plane or a disk) to be an arc of a circle: the unique circle that passes through both points and meets the boundary (of the half-plane or disk) at right angles.

`\hypergeodesic{$z_1$,$z_2$}`
`\hypergeodesic*{$z_1$,$z_2$}`

This produces the hyperbolic geodesic from $z_1 = (x_1,y_1)$ to $z_2 = (x_2,y_2)$. For guaranteed results both points should be in the unit disk (i.e., $|z_1| < 1$ and $|z_2| < 1$) or, for the star form, both in the upper half-plane (i.e., $y_1 > 0$ and $y_2 > 0$). However, if these are not satisfied an arc of the circle described above will be drawn whenever not prevented by a numeric overflow. The star form produces the geodesic for points in the upper half-plane. The normal form produces the geodesic for points in the unit disk (inside of the circle of radius 1 and center $(0,0)$). One can use transforms of these to get geodesics suitable for other disks and other half-planes.

4.2.5 CURVES

`\curve[⟨tension⟩]{⟨$p_0$⟩,⟨$p_1$⟩,...}`
`\cyclic[⟨tension⟩]{⟨$p_0$⟩,⟨$p_1$⟩,...}`
`\closedcurve[⟨tension⟩]{⟨$p_0$⟩,⟨$p_1$⟩,...}`

These figure macros produce 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 `\cyclic` variant arranges for the last point to be connected (smoothly) to the first, and produces a closed METAFONT Bézier curve. The command `\closedcurve` is an alias for `\cyclic`.

The optional ⟨*tension*⟩ 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{⟨num⟩}`

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

Sometimes one would like a convex set of points to produce a convex curve. This will not always

be the case with \curve or \cyclic. You can verify this with the following example, where the list of points traces a rectangle:

    \cyclic{(0,0),(0,1),(1,1),(2,1),(2,0),(0,0)}

To produce a convex curve, use one of the following:

\convexcurve[$\langle tension\rangle$]{$\langle p_0\rangle$,$\langle p_1\rangle$,...}
\convexcyclic[$\langle tension\rangle$]{$\langle p_0\rangle$,$\langle p_1\rangle$,...}
\closedconvexcurve[$\langle tension\rangle$]{$\langle p_0\rangle$,$\langle p_1\rangle$,...}

These figure macros can be used even if the list of points is not convex, and the result will be convex where possible. The third one is an alias for for the second one.

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

\fcncurve[$\langle tension\rangle$]{(x_0,y_0),(x_1,y_1),...}

This figure macro produces 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 $\langle tension\rangle$ is a number greater than $1/3$ 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 that set with \settension, initially 1. 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 actually be any positive number, but only values greater than $1/3$ guarantee the property of never doubling back.

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

This figure macro produces a a sequence of line segments starting from the point $\langle p_0\rangle$, and extending along the (2-dimensional vector) displacement $\langle v_1\rangle$. The next segment is from the previous segment's endpoint, along displacement $\langle v_2\rangle$. This continues for all listed displacements, a process similar to 'turtle graphics'.

4.2.6   BAR CHARTS AND PIE CHARTS

\barchart[$\langle start\rangle$,$\langle sep\rangle$,$\langle r\rangle$]{$\langle h\text{-}or\text{-}v\rangle$}{$\langle list\rangle$}
\bargraph...
\gantt...
\histogram...
\mfpbarchart...
\mfpbargraph...
\mfpgantt...
\mfphistogram...

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

Since \barchart never draws anything, there is no particular reason it needs to be inside an mfpic environment. Starting with version 0.9 of MFPIC this is no longer required, but the command

name `\mfpbarchart` must be used outside (in case some other package also defines `\barchart`). One can use any of the four synonyms listed that start with '`\mfp`'. The commands to draw the bars are still required to be inside an `mfpic` environment.

⟨*h-or-v*⟩ 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. ⟨*list*⟩ should be a comma-separated list of numbers and ordered pairs giving 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. ⟨*start*⟩ is the *y*-coordinate of the bottom edge of the first bar, ⟨*sep*⟩ is the distance between the bottom edges of successive bars, and ⟨*r*⟩ is the fraction of ⟨*sep*⟩ 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 ⟨*sep*⟩ negative, and making ⟨*start*⟩ the top edge of the first bar.

The fraction ⟨*r*⟩ 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 on top of 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.

`\chartbar{`⟨*num*⟩`}`
`\graphbar{`⟨*num*⟩`}`
`\histobar{`⟨*num*⟩`}`
`\ganttbar{`⟨*num*⟩`}`

The figure macro `\chartbar` (synonyms `\graphbar`, `\ganttbar`, and `\histobar`) takes a number from 1 to the number of elements in the list of data of the most recent `\barchart` command and produces the corresponding rectangular path computed by that command. This behaves just like any other figure macro, and the prefix macros from section 4.5 may be used to give adjacent bars contrasting colors, fills, etc.

`\piechart[`⟨*dir*⟩⟨*angle*⟩`]{`⟨*c*⟩`,`⟨*r*⟩`}{`⟨*list*⟩`}`
`\mfppiechart...`

The macro `\piechart` also does not draw anything, but computes the `\piewedge` regions described below. The first part of the optional parameter, ⟨*dir*⟩, is a single letter to indicate a direction: '`c`' for *clockwise* or '`a`' for *anticlockwise*. The ⟨*angle*⟩ is the angle in degrees of the starting edge of the first wedge. The defaults correspond to `[c90]`, which means the first wedge starts at 12 o'clock and proceeds clockwise.

The first required argument contains the center ⟨*c*⟩ and radius ⟨*r*⟩ of the chart. The second required argument is the list of data: positive numbers separated by commas.

Since this command never actually draws anything, only defining the wedges, it makes sense to heave it available outside the drawing environment. Starting with version 0.9 of MFPIC that is the case, but the command name is `\mfppiechart` (to avoid a name clash with some other package's

\piechart command). The command to draw wedges (\piewedge, see below) is still required to
be inside an mfpic environment.

\piewedge[⟨*spec*⟩⟨*trans*⟩]{⟨*num*⟩}

This figure macro takes a number from 1 to the number of elements in the list of data of the most
recent \piechart command and produces the corresponding wedge-shaped path computed by that
command. By default, the path is positioned as computed by that \piechart command, but 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.

4.2.7 BRACES

This figure is intended to group some graphical objects and label them.

\gbrace{⟨z_1⟩,⟨*C*⟩,⟨z_2⟩}

This figure macro creates the shape of a brace (i.e., a '}') with its ends at ⟨z_1⟩ and ⟨z_2⟩ and
its 'center' cusp at ⟨*C*⟩. The three points must be expressed as ordered pairs or as METAFONT pair
expressions, and must be separated by commas. The 'width' of the brace (the distance from ⟨*C*⟩ to
the line through the other two points) is computed automatically and should not be 0. The cusp of
the brace will not necessarily be in the center of the brace. Users position it with their choice of ⟨*C*⟩.
The cusp should not be positioned too close to one of the endpoints as this can distort the brace.

4.3 Colors in MFPIC.

4.3.1 METAPOST COLOR FUNCTIONS

Because of changes to color handling with METAPOST 1.000, we will have to give two descriptions
of some operations. For brevity, we will refer to METAPOST versions before the addition of the
cmykcolor data type as 'early' METAPOST and the versions afterward as 'recent' METAPOST. Early
METAPOST actually ended with version 0.642. When development resumed, beta test versions began
with 0.900. Any version 0.900 or later qualifies as 'recent'.

In early METAPOST, the only color data type is a triple of numbers like (1,.5,.5), with the
components between 0 and 1, representing red, green and blue levels, respectively. White is given
by (1,1,1) and black by (0,0,0). Recent METAPOST has the color data type (refered to as ei-
ther color or rgbcolor) as well as the cmykcolor type. A cmykcolor is a quadruple of numbers like
(1,.2,0,.3), with components between 0 and 1 representing levels of cyan, magenta, yellow and
black. White is represented by (0,0,0,0). While black can be obtained in several ways,(0,0,0,1)
is the simplest.

METAPOST also has color variables (and cmykcolor variables) and several have been predefined.
The colors red, green, blue, white and black are built in to METAPOST and are of type rgbcolor.
Colors cyan, magenta and yellow are defined by MFPIC's METAPOST support macros to be cmyk-
color. In addition, MFPIC defines grayscaleblack, grayscalewhite, cmykblack, cmykwhite,
rgbblack and rgbwhite. These give black and white in the indicated data type (grayscale being a
numeric: 0 for black, 1 for white).

All the names in the LATEX COLOR package's dvipsnam.def have also been predefined by
MFPIC as color variable names. Since METAPOST allows color expressions, colors may be added

together (as long as they are the same type) and multiplied by numerics. Multiplication by a number between 0 and 1 darkens a rgbcolor, but lightens a cmykcolor.

Moreover, several METAPOST color functions have been defined in `grafbase.mp`. These have the same names as the color models. Strictly speaking, it is never necessary to use these in recent METAPOST. However, since METAFONT and early METAPOST don't have a data type consisting of quadruples, and METAFONT doesn't have one for triples, these functions allow the same MFPIC code to be used for all three figure processors. These functions are defined to convert to a usable data type, (which may be ignored in METAFONT).

cmyk(c,m,y,k)

In early METAPOST, this 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 rgb equivalent of cyan. In recent METAPOST this produces the cmykcolor with the given components. That is, cmyk(1,0,0,0) simply produces $(1,0,0,0)$, the cmyk coding for cyan.

gray(g)

In early METAPOST, this converts a numeric g (designating a level of gray) to the corresponding multiple of white: (g,g,g). In recent METAPOST, commands to draw paths or pictures in a particular color will accept a numeric parameter instead of color or cmykcolor, so in recent METAPOST this command simply returns the given numeric g.

named($\langle name\rangle$), rgb(r,g,b)

These are essentially no-ops. However; rgb() will truncate the arguments to the 0–1 range, and set an unknown argument to 0. An unknown $\langle name\rangle$ is converted to black (in the appropriate color model if $\langle name\rangle$ is an unknown color variable, otherwise rgb black).

RGB(R,G,B)

Converts an RGB color specification to rgb. It divides each component by 255, and performs the same truncations as rgb(). The RGB model consists of a triple of numbers between 0 and 255. Originally, the model required they be integers. However, since they are converted to fractions anyway, it doesn't matter in this command.

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

```
\draw[0.5*RGB(255,0,0)+0.5*cmyk(1,0,0,0)]\circle{(0,0),1}
```

to have a circle drawn in a color halfway between red and cyan (which turns out to be the same as gray(0.5)). In recent METAPOST, however, this would be an error, as one cannot add two different data types (rgbcolor and cmykcolor). So MFPIC supplies conversion functions.

makecmyk $\langle clr\rangle$
makergb $\langle clr\rangle$
makegray $\langle clr\rangle$

In recent METAPOST, the $\langle clr\rangle$ can be a known color name, a constant of type numeric, rgbcolor, or cmykcolor, or the result of a color function. Then makecmyk returns the cmykcolor equivalent, and makergb returns the rgbcolor equivalent (a numeric $\langle clr\rangle$ is interpreted as a grayscale color). Unknown colors produce a black in the appropriate model. Then one can use

```
\draw{.5*RGB(255,0,0) + .5*makergb cmyk(1,0,0,0)}\circle{(0,0),1}
```

If one has forgotton whether RGB returns an rgbcolor, one could write makergb RGB(255,0,0) to be sure to get an rgbcolor.

The first two commands are never necessary in early METAPOST, but they are still defined: they simply return the given color if it is a known argument of type color, or apply the function gray() is it is numeric, and return black for an unknown name.

The last one makegray converts any color to a numeric, and then returns either that number (recent METAPOST) or that multiple of white (early METAPOST). In METAFONT, all three pass the (presumably numeric) argument ⟨*clr*⟩ unchanged.

All three functions return some version of black if ⟨*clr*⟩ is not a color of some type, or has an unknown value.

4.3.2   ESTABLISHING MFPIC DEFAULT COLORS

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

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, and \plotsymbol commands, as well as \gridpoints and \plrgridpoints, use \pointcolor, and for arrowheads, \headcolor. One can set the color used by \gclear with \backgroundcolor (the same color will also be used in the interior of unfilled points that are drawn with \point) and, when mplabels is in effect, the color of labels can be set with \tlabelcolor.

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 named background). Thus, after drawcolor has been set, one can issue the command \fillcolor{drawcolor} to fill with the same color.

As previously discussed, all these colors are initially set to black except background is set to white.

If the optional ⟨*model*⟩ argument is omitted, the color specification may be any expression recognized as a color by METAPOST. It is highly recommended (for portability) that one use either a predefined name or one of the color functions of the previous section.

When the optional ⟨*model*⟩ is specified in the color setting commands, it determines the format of the color specification as in figure 1.

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 0 indicating black, 1 white.
named	A METAPOST color variable name either predefined by MFPIC or by the user.

Figure 1: Color specifications

MFPIC translates the command:

```
\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. One sees from this that the optional argument is never necessary. It's there only to make the LaTeX user comfortable.

4.3.3 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 any LaTeX package (such as COLOR or XCOLOR). In particular a color defined only by LaTeX's \definecolor command will remain unknown to MFPIC. Conversely, LaTeX commands will not recognize any color defined only by \mfpdefinecolor.

4.3.4 METAFONT COLORS

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 not 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 4.5.2) 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 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 4.5.2 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 erases all affected pixels before adding the dots to simulate gray. In particular, \gfill[white] should have the same effect as \gclear.

4.4 Modifying the figures.

Some MFPIC macros operate by *modifying* a figure macro: if you want to turn an open arc into a closed figure by adding a straight line, you can write:

```
    \lclosed\arc{(0,0),(1,0),45}.
```

These are always prefixed to some figure drawing command, and apply only to the next following figure macro provided that only other prefix commands intervene. This is a rather long section, but even more modification prefixes are documented in subsection 4.10.2.

The combination of a modifying macro, followed by a figure macro, can usually be thought of as a new figure macro, to which further prefixes might be prepended.

More precisely: all prefix macros have an *input* path, an *output* path, and a *side effect*. The input is the path that is output by the *following* prefix or figure macro. The output is either the same as the input or a modification of it. The side effect might be a drawing or filling of the path or the addition of an arrowhead.

We list here a classifications of prefix and figure macros that is useful for understanding the MFPIC system.

Figure macros. These take no input path; they must come last in a sequence. They output the path they were designed to produce. Examples are \circle, \rect and \polygon. If they have no prefixes, or are preceded only by appending macros (see next), they invoke a default rendering of the path (usually a drawing as a solid stroke) as the side effect.

Appending macros These pass their input unchanged as their output. Their side effect is the appending of some object such as an arrow head or tail. Currently only the various prefix macros whose names begin with arrow are appending macros (see subsection 4.4.3). But \reverse, which technically modifies a path and has no side effect, is coded as an appending macro so that it will work correctly with arrows. Think of it as 'appending' a new direction.

Rendering macros These pass their input unchanged as their output. They have the side effect of adding or subtracting ink from a picture in the shape of the input path. Examples are \draw, \dotted, \gfill and \gclip.

Modifying macros These output the result of applying their intended modification to the input path. Examples are macros that close the path if it was open, macros that apply a transformation such as a rotation, and macros that return only a part of a path. If they have no prefixes, or are preceded only by appending macros (see above), they also invoke a default rendering of the output path (usually a drawing as a solid stroke of the modified path) as the side effect.

4.4.1 CLOSURE OF PATHS

It should be pointed out that the closure macros will leave already closed paths unchanged, so it is always safe to add one when uncertain. Moreover, if the path is not closed but the endpoints are identical, \lclosed and \bclosed will close it without adding any path segment.

```
\lclosed...
\bclosed[⟨tens⟩]...
\sclosed[⟨tens⟩]...
```

These modifying macros all turn an open path into a closed one. If the path is already closed, they do nothing.

\lclosed makes an open path into a closed path by adding a line segment between the endpoints of the path. In the special case where the path ends exactly where it begins, all \lclosed does is change the type of the path from open to closed.

The \bclosed 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 $\langle$*tens*$\rangle$ greater than 1, the default. The command \settension (see subsection 4.2.5) can be used to change the default.

\sclosed 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 a result almost identical to \cyclic given the same points and tension values. The optional tension argument is as in the \bclosed command.

There are two other closure commands but, because they are associated with particular types of paths (splines), we delay their discussion until those are discussed (subsection 4.12.1).

\makesector\arc[$\langle$*fmt*$\rangle$]{$\langle$*spec*$\rangle$}

The modifying macro \makesector can be applied to any path, but its name makes sense (and its action is predictable) only if that path is an arc. It appends line segments from the center of the arc's circle to the ends of the arc, producing a closed path. It is useful if one doesn't know where the center of the arc is (a required parameter of \sector). It works by selecting the first point, a middle point, and the last point of the following path, then calculates the center of the circle through those three points.

4.4.2   REVERSAL, CONNECTION AND OTHER PATH MODIFICATIONS

\reverse...
\reversepath...

This modifies the following path by reversing its sense. This will affect the direction of arrows: bi-directional arrows can be coded with \arrow\reverse\arrow..., where the leftmost \arrow prefix applies to the *reversed* path. The order of endpoints for a connect environment will also be affected. The command \reversepath is exactly the same. It has been added (in vresion 1.10) to more closely match the names of other modification macros (see subsection 4.10.2).

\connect ... \endconnect

The macro \connect produces a connected path by joining all the paths following it up to the matching \endconnect command. Line segments are added from the end of one path to the start of the next. The whole group acts as one figure macro, permitting any prefix macros to come before.

In LaTeX, instead of this pair of macros, an environment named connect may be used. For example

```
    \lclosed
    \begin{connect}
      \curve{(2,1),(1,2),(0,1)}
      \polyline{(0,0),(2,0)}
    \end{connect}
```

produces a closed figure consisting of one smooth curve and three line segments: the segment produced by \polyline, the segment added by the connect environment, and the segment added by \lclosed.

`\partpath{`⟨*frac1*⟩`,`⟨*frac2*⟩`}`...
`\subpath{`⟨*num1*⟩`,`⟨*num2*⟩`}`...
`\trimpath{`⟨*dim*$_1$⟩`,`⟨*dim*$_2$⟩`}`...
`\trimpath{`⟨*dim*⟩`}`...

These macros modify the following path by producing only a part of it. 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 the fraction ⟨*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.

The `\trimpath` macro takes two dimensions separated by commas and trims those lengths off the initial and terminal ends of the following path. Alternatively, it takes one dimension and and trims that length off of both ends. If any of ⟨*dim*$_1$⟩, ⟨*dim*$_2$⟩ or ⟨*dim*⟩ is missing, it is taken to be `0pt`. This works by finding the points of intersection between the path and circles around the endpoints with the given dimensions as radii. If the path is shorter than either dimension, it will not intersect either circle and nothing will be trimmed. Similar problems can occur, at one end or the other, if the path is shorter than the sum of the dimensions.

`\parallelpath{`⟨*dist*⟩`}`...

This modifying macro takes the following path and returns a path that follows beside it, keeping a fixed distance ⟨*dist*⟩ to the left. If ⟨*dist*⟩ is negative, it keeps to the right. Left or right is from the point of view of a traveller following the given path from start to finish. The distance is a pure number in *graph* coordinates. Note: this should be compared to the first optional argument of `\doubledraw` (see subsection 4.5.1), which requires an absolute dimension like `2pt`, even though it is implemented using the internal code of `\parallelpath`.

The calculation of the parallel path is approximate and rather inefficient. It is likely to produce inexplicable small loops where it tries to follow the inside of tight turns (radius less than ⟨*dist*⟩). Actual corners, (which might be thought of as turns of radius 0) are usually detected and dealt with in a reasonable manner. However, if the path is made up of segments of length ⟨*dist*⟩ or less, this is unlikely to work correctly at all.

`\arccomplement`...

This macro, to work properly, must be followed by an arc of a circle. It produces the complementary arc. That is, it produces the circular arc, which would, if appended to the following arc, complete the circle. The complementary arc will have the same direction, clockwise or anticlockwise, as the original. The arc that follows doesn't have to be produced by `\arc`, as in the following example:

```
\draw[blue]\arccomplement
  \draw[red]\partpath{0,.333}
  \circle{(0,0),1}
```

This will draw 1/3 of this circle in red and the rest of it in blue.

METAFONT cannot check if a path is really a circular arc. The METAFONT code, like that of `\makesector` (see subsection 4.4.1), selects three key points on the arc, then it produces the rest of the circle much the same way as the internal code of `\arc[t]` (the three point option for `\arc`). Thus, it will produce *some* arc from the end of any following path to its beginning (or a straight line if the three chosen points happen to lie in a straight line). However, the result needn't bear any significant relation to the original path.

`\interpolatepath{`⟨*frac*⟩`,`⟨*path1*⟩`}`...

This modifying macro takes the following path (call it ⟨*path0*⟩) and computes a new path that is ⟨*frac*⟩ of the way "between" it and the argument ⟨*path1*⟩. The argument ⟨*path1*⟩ would usually be the name of a METAFONT path variable used to store a figure (see `\store` from subsection 4.10.2). However it can actually be any legal METAFONT path expression. The argument ⟨*frac*⟩ is a number. If ⟨*frac*⟩ is 0, nothing is done and the following path ⟨*path0*⟩ is produced; if ⟨*frac*⟩ is 1, then the argument ⟨*path1*⟩ is produced. For values of ⟨*frac*⟩ between 0 and 1 the resulting path is somewhere between the two. Numbers larger than 1 or less than 0 produce an extrapolated path. An ordered pair can be supplied instead of the argumant ⟨*path1*⟩: it will be interpreted as a trivial path. If ⟨*path0*⟩ (the following figure) is closed and if ⟨*path1*⟩ is an ordered pair or a closed path, then the resulting path will also be closed. Otherwise it will not be.

What this command actually interpolates are the key points of the paths and the directions of travel at those key points. Therefore, if the two paths are visually very similar but have very different node structure, then the interpolated path can be quite unexpectedly different from both of them. For example

```
\store{ABC}\circle{(0,0),1}
\interpolatepath{.5,ABC}\reverse\circle{(0,0),1}
```

produces a straight line from $(1,0)$ to $(-1,0)$ (and back again).

4.4.3   ARROWS

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

This macro adds 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 (for example) 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, it is set back in the direction after the rotation. The optional ⟨*color*⟩ defaults to `headcolor`, initially black. The optional parameters may appear in any order, the indicated key character determining the meaning of a parameter. The key letter `l` for 'length' can be replaced by `s` for 'size'.

There is also a star-form: If `\arrow` is called as `\arrow*`, then any part of the tip of the following curve that lies outside the arrowhead shape is clipped off. Imagine a rectangle with one side connecting the ends of the barbs and the opposite side passing through the tip. Everything in that rectangle outside the arrowhead is erased, so be careful using this (also see comments about META-POST's method of 'erasing' in the description of `\gclear` in \subsection 4.5.2). One use of this is adding an arrowhead to a figure rendered with `\doubledraw` (see the next section) or with a rather large pen diameter (see section 4.11).

For the star-form to work, the head has to be added after the path is drawn. What this means in practice is that the `\arrow*` command must come before any drawing command in the list of prefixes. This is because prefix macros add their elements to the result of everything that follows. If you `\store` a curve in a path variable (see subsection 4.10.2), and draw the path and the arrowhead in separate commands, then the arrow command must come *after* the drawing command.

\arrowhead{⟨*symbol*⟩}[l⟨*length*⟩][r⟨*rotate*⟩][b⟨*backset*⟩][c⟨*color*⟩]...
\arrowmid{⟨*symbol*⟩}[l⟨*length*⟩][r⟨*rotate*⟩][f⟨*fraction*⟩][c⟨*color*⟩]...
\arrowtail{⟨*symbol*⟩}[l⟨*length*⟩][r⟨*rotate*⟩][f⟨*forward*⟩][c⟨*color*⟩]...

These macros add some sort of symbol at different locations along a path. The first adds an arrowhead, but the head can be any appropriately designed symbol. It has been arranged that any of the symbols usable in \plotsymbol (see subsection 4.2.1) can be used: you can have Diamond- or Asterisk-tipped arrows. The special symbol Arrowhead produces the same shape as the head in the \arrow command. In total eight special ⟨*symbols*⟩ have been made available, intended for use with \arrowhead, \arrowmid and \arrowtail. Here is a list and description of all these symbols.

Arrowhead The shape that would be drawn at the end of a path by \arrow.
Leftharpoon The left half of Arrowhead.
Rightharpoon The right half of Arrowhead.
Crossbar A short line crossing the path perpendicularly unless rotated.
Leftbar Essentially the left half of Crossbar.
Rightbar The right half.
Lefthook An open semicircle with its open face in the direction of the path, added to the left side of the path.
Righthook Like Lefthook but on the right side.

Here 'left' and 'right' are from the point of view of an observer facing in the direction of the path.

If the symbol is a closed path (see subsection 4.4.1 for the difference between a closed path and one that merely looks closed), the head will be filled, otherwise its outline will be drawn. Thus \arrowhead{Diamond} draws an outline, and \arrowhead{SolidDiamond} draws a filled shape because Diamond has been left open, while SolidDiamond has been defined to be closed.

It is possible, to get an outline drawn with the inside erased: just place the solid version with color background (usually the same as white) and then the outline version. This can produce a pleasing result. But recall that the prefix macro nearest the figure macro is executed first. For example:

    \arrowmid{Circle}\arrowmid{SolidCircle}[cwhite]\polyline{(0,0),(1,1)}

The symbol is always rotated so that it points in the direction of the path (for this purpose, all symbols are initially assumed to point straight upward) before the [r⟨*rotate*⟩] parameter is applied.

There is a star-form \arrowhead* that behaves like \arrow* (when possible). The optional arguments are exactly as in \arrow, with the same defaults for all of them.

The second command, \arrowmid, places the symbol somewhere between the start and the end of the path. In this case the optional parameter [f⟨*fraction*⟩] gives the location of the symbol as a fraction of the length of the path. The default is [f0.5], which places it approximately in the middle. The other optional arguments have the same meaning as for \arrowhead. As with \arrowhead, the symbol is rotated to 'point' in the direction of the path before the [r⟨*rotate*⟩] is applied.

The third command \arrowtail places the symbol at the start of the path. Otherwise it behaves as the other two commands, except the option [f⟨*forward*⟩] is an amount to shift the symbol forward from that first point.

One might be tempted to use \arrowmid with the ⟨*fraction*⟩ equal to 1 or 0 to get arrowheads or tails. This will work sometimes. However, some shapes have a 'tip', that is, a particular point designated as the tip of the arrowhead. The \arrowhead and \arrowtail commands pay attention to this, while \arrowmid does not. Also, \arrowmid has no star-form.

You can design your own ⟨*symbol*⟩ for these commands: use \store to store a path in a path variable (see subsection 4.10.2). These commands assume that the length is 1, that the symbol 'points' up and that the 'tip' (the 'pointy end') is at $(0,0)$ (unless the pair variable ⟨*symbol*⟩.tip is defined, in which case that is taken to be the tip). So draw your symbol pointing up with its tip at $(0,0)$ and

its length equal to 1 (graph unit). For example the following produces a solid head with a common shape:

```
\store{myAH}\polygon{(-.5,-1)(0,0),(0.5,-1),(0,-.7)}
\arrowhead{myAH}\arc{(-10,0),(10,0),90}
```

If you replace the `\polygon` above with `\polyline`:

```
\store{myAH}\polyline{(-.5,-1)(0,0),(0.5,-1),(0,-.7),(-.5,-1)}
```

the path will not be closed and so the arrowhead will not be filled in.

To make the star-form work with such self-defined symbols, one must also define a closed path `myAH.clear` that gives the region to be erased. In the above example:

```
\store{myAH.clear}\polygon{(-.5,-1),(-.5,0),(.5,0),(.5,-1),(0,-.7)}
```

4.5 Rendering figures.

When MFPIC is loaded, the initial way in which figures are drawn is with a solid outline. That is, `\polyline{(1,0),(1,1),(0,0)}` will draw two solid lines connecting the points. It is possible to establish a different default (see `\setrender` in subsection 4.5.3), however that default is used only when no explicit rendering prefix is present. That is, when the macros in this section are used, any previously established default is overridden.

`\norender`...

This causes the following path not to be rendered at all. This can be used to override MFPIC's automatic rendering rules. See section 4.10.2, page 59 for an example where one might need to do this.

4.5.1 DRAWING

`\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`, `\doubledraw`, `\plot`, `\plotnodes`, and `\gendashed`,

`\doubledraw[`⟨*sep*⟩`][`⟨*color*⟩`]`...

This rendering macro draws the path with a double line. The default separation (distance between centers of the two penstrokes) is twice the pen diameter. This normally leaves one line thickness of white space between. You can change this with the [⟨*sep*⟩] argument. In order to make the space between the lines transparent, this command is implemented by calculating two curves that are parallel to the given curve and drawing those. For technical reasons, that calculation is rather lengthy so this is somewhat inefficient and users of slow machines might want to avoid it. See also comments at `\parallelpath` in subsection 4.4.2.

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

This rendering macro draws dashed segments along the path specified. 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 $1/n$ 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*⟩]...

   This rendering macro 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, this rendering macro draws copies of ⟨*symbol*⟩ along the path. Possible symbols are those listed under \plotsymbol in subsection 4.2.1. The default ⟨*size*⟩ is \pointsize (initially `2pt`) and the default ⟨*space*⟩ is \symbolspace (initially `5pt`).

\plotnodes[⟨*size*⟩]{⟨*symbol*⟩}...

   This rendering macro places a symbol at each *node* of the path that follows. Possible symbols are those listed under \plotsymbol in subsection 4.2.1. 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 (subsection 4.6.3), each of the data points in the file is a node. In the function macros (subsection 4.6.2) 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, MFPIC offers a generalized dashing command. Before using it, one must first establish a named dashing pattern with the above command. The ⟨*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 spaces 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.[13]

   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 figure macro 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

---

[13]Experienced METAPOST users could also set the `linecap` variable to `butt`.

dash pattern are 0, \gendashed responds by drawing a solid curve. The same is true if the pattern has only one entry.

\zigzag{⟨*start*⟩,⟨*end*⟩,⟨*wl*⟩,⟨*amp*⟩}...
\sinewave[⟨*tens*⟩]{⟨*start*⟩,⟨*end*⟩,⟨*wl*⟩,⟨*amp*⟩}...
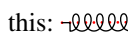
These figure macros both draw a solid line that crosses from one side of the path to the other. The \zigzag makes a jagged result while the \sinewave makes a smooth one. The optional argument of \sinewave is a 'tension' and controls how smooth the result is. The default tension is 1. Higher values make a less smooth path, and values of 10 or so produce a result almost indistinguishable from \zigzag. Tension is required to be greater than 3/4.

The mandatory arguments consists of four dimensions separated by a comma. The rendering produced by these macros actually follow the path a little way at the start and end of the path. This is controlled by the dimensions ⟨*start*⟩ and ⟨*end*⟩.

The third dimension, ⟨*wl*⟩, is the distance from one 'peak' to the next (the 'wavelength'). The second, ⟨*amp*⟩, is the maximum distance to either side of the true path (the 'amplitude'). Reasonable values of ⟨*wl*⟩ and ⟨*amp*⟩ are 8pt and 2pt, respectively. These proportions (4 to 1) causes the zigzag and the sinewave to cross the path at an angle of about 45 degrees, a rather pleasant result. Those sizes are close to optimal: too much smaller and the rendering just looks like a fuzzy line, too much larger, and bends in the path will distort the zigzagging.

The zigzags zig to the left first if ⟨*amp*⟩ is positive, to the right if it is negative. For closed curves, the beginning and end are constructed to meet smoothly. It is always arranged that there are an equal number of left zigs and right zags, so the ⟨*wl*⟩ is only approximate.

\corkscrew[⟨*tens*⟩]{⟨*start*⟩,⟨*end*⟩,⟨*wl*⟩,⟨*amp*⟩}...
\coil[⟨*tens*⟩]{⟨*start*⟩,⟨*end*⟩,⟨*wl*⟩,⟨*amp*⟩}...

This rendering macro draws a coil or corkscrew that coils around a given path, something like this: ⌐ℓℓℓℓℓ⌐ (the red dots show the actual path). The ⟨*tens*⟩ is a tension option that controls how 'loopy' the result will be (the higher the number the more jagged). The mandatory argument contains four explicit dimensions. The first two, ⟨*start*⟩ and ⟨*end*⟩ are as in \zigzag. The ⟨*wl*⟩ is the distance from one loop to the next, and ⟨*amp*⟩ is the distance from the true path to the tops (or bottoms) of the loops. If ⟨*amp*⟩ is positive, the tip of the loop is to the left of the path, if negative it is to the right. The example at the start of this paragraph was drawn using the following code:

```
\mfpic{0}{33}{0}{6.4}
\dotsize=1pt
\drawcolor{red}
  \dotted\polyline{(0,3.2),(33,3.2)}
\drawcolor{black}
  \coil[1.5]{3pt,3pt,4.8pt,3.2pt}\polyline{(0,3.2),(33,3.2)}
\endmfpic
```

4.5.2 Shading, filling, erasing, clipping, hatching

For the purposes of this section, 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 might be open and may need to be explicitly closed, for example by \lclosed. The (already) closed paths are those that have 'closed' or 'cyclic' in their name plus:

\belowfcn, \border, \btwnfcn, \btwnplrfcn, \chartbar (and its aliases),
\circle, \ellipse, \fullellipse, \levelcurve, \makesector,\piewedge,
\plrregion, \polygon, \pshcircle. \rect, \regpolygon, \sector, \tlabelcircle,
\tlabelellipse, \tlabeloval, and \tlabelrect.

The macros of this section 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. Note that these macros override the default rendering, so if you want some sort of fill pattern *and* an outline drawn, you need an explicit prefix for both.

\gfill[⟨*color*⟩]...

This rendering macro 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...

This rendering macro 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. However, if an mfpic environment utilzes *background text* (see subsection 4.7.1), part of the background text may appear to be 'erased'. Unfortunately, there is little that can be done about this.

\gclip...

This rendering macro erases everything *outside* the subsequent closed path from the picture (except text labels under some circumstances, see section 2.2 and 2.3). Note that this is a true erasing, even in METAPOST.

\shade[⟨*shadesp*⟩]...

This rendering macro 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 0.5bp. The optional argument specifies the spacing between (the centers of) the dots, which defaults to the TEX dimension \shadespace, initially 1pt. If shadewd is larger than \shadespace, 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 about 78% of white.[14] 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*⟩]...

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

---

[14]If \shadewd is w and \shadespace is s, then the level of gray is $1 - (.88w/s)^2$, where 0 denotes black and 1 white.

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 5bp. The dots are colored with fillcolor. In METAFONT, nonblack values of fillcolor will produce shaded dots.

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

This rendering macro 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 or comma).

An angle of 0 yields horizontal lines, nonzero angles indicate rotations from horizontal where, as usual, positive angles indicate anticlockwise rotation.

\lhatch[⟨*hatchsp*⟩][⟨*color*⟩]...
\rhatch[⟨*hatchsp*⟩][⟨*color*⟩]...
\hatch[⟨*hatchsp*⟩][⟨*color*⟩]...
\xhatch[⟨*hatchsp*⟩][⟨*color*⟩]...

These rendering macros are just \thatch with predefined values of the angle. \lhatch fills the region with left slanted lines (from upper left to lower right). It is exactly the same as

\thatch[⟨*hatchsp*⟩,-45][⟨*color*⟩]...

\rhatch draws right slanted lines (lower left to upper right). It is exactly the same as

\thatch[⟨*hatchsp*⟩,45][⟨*color*⟩]...

\hatch (\xhatch is a synonym) draws lines 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). However, hatching on top of another filling macro is a common way to fill in *two* regions in such a way that the overlap area is clearly evident. Hatching is at least less garish than polkadots.

\gradient{⟨*clrfcn*⟩,⟨*width*⟩,⟨*angle*⟩}...

Neither METAPOST nor METAFONT can do true gradients, but this rendering macro obtains a good approximation by filling adjacent narrow strips in a range of colors. The argument ⟨*width*⟩ is the width of those strips, and it should be specified in absolute units, ⟨*angle*⟩ is the angle of these strips (horizontal being 0 degrees). The first argument takes a little explanation.

The ⟨*clrfcn*⟩ should be the name of a function, say gr, such that gr(t) returns a color[15] for any value of *t* from 0 to 1 (inclusive). Such a function can be defined with \fdef (see subsection 4.6.1). The first strip will have the color returned for t = 0 and the last will have the color returned for t = 1. One kind of gradient fill is obtained by a simple interpolation between two colors in the same model:

```
\fdef{gr}{t}{(1-t)*red + t*blue}
\gradient{gr,3pt,45}\circle{(0,0),1}
```

---

[15]It can also return a number between 0 and 1, which will result in shades of gray.

This example will start the gradient with red and end it with blue. For an angle of 0 the starting color is at the bottom and the ending color at the top, for other angles simply rotate that description. The above circle will be red at the lower right and blue at the upper left.

This type of gradient is called an *axial* gradient. The following is another kind, based on a color function of *two* variables over an area.

\areagradient{⟨*clrfcn*⟩,⟨*h-dim*⟩,⟨*v-dim*⟩}...

Instead of filling with strips of different colors, \areagradient fills with "pixels" of different colors. These are rectangles that have size ⟨*h-dim*⟩ by ⟨*v-dim*⟩, which values must be specified in absolute units. These rectangles are filled with the color determined by ⟨*clrfcn*⟩. This must be a function of *two* parameters that returns a color for values of these parameters from 0 to 1 (inclusive). For example,

```
\fdef{agr}{t,u}%
    {(1-t)(1-u)*white + (1-t)*u*red + t*(1-u)*green + t*u*blue}
\gradient{agr,3pt,4pt}\rect{(0,0),(1,2)}
```

The color returned for $(0,0)$ is at the lower left and the color returned for $(1,1)$ is at the upper right. In the above example, the rectangle will be white at the lower left, red at the upper left, green at the lower right and blue at the upper right.

Our last gradient is something like the first, but in polar coordinates. The colors vary with the distance from a center point.

\radialgradient{⟨*clrfcn*⟩,⟨*width*⟩,⟨*center*⟩}...

This gradient fills with concentric circular strips whose center is ⟨*center*⟩ and whose thickness is ⟨*width*⟩. The ⟨*clrfcn*⟩ is as in \gradient. The circle of radius ⟨*width*⟩ and center ⟨*center*⟩ is filled with the color returned for parameter value 0. The largest concentric circular strip is filled with the color returned for parameter value 1.

These commands all initially compute a bounding figure for the curve. In the first case it is a rotated rectangle, in the second case an upright rectangle, and in the third case a circle centered at the given point. The interiors of the rectangles are considered to have coordinates (t, u) that vary from $(0,0)$ at the lower left to $(1,1)$ at the upper right. The inside of the circle is considered to have the polar coordinate *r* ranging from 0 at the center to 1 at the boundary. The relevant coordinate(s) are fed to the specified ⟨*clrfcn*⟩ and the returned color is used to fill the relevant portion of the rectangle or circle. The whole picture is then clipped to the boundaries of the given closed curve and the result added to the picture.

The process is somewhat wasteful of memory in METAPOST, as each strip or pixel's path is kept in memory and written to the output file. This can be quite large for \areagradient if the pixel dimensions are too small. For example, covering a one inch square with pixels 1 point on each side takes over 5000 paths and the resulting EPS is over 100,000 bytes in size. I would recommend dimensions on the order of 2 to 3 points. Larger dimensions are not as visually appealing, and smaller dimensions waste memory with little improvement in appearence.

This command works in METAFONT using a ⟨*clrfcn*⟩ that returns numeric values in the range 0 to 1. The result is much like \gfill[⟨*clr*⟩] (see the beginning of this section) except the dots simulating a gray fill will vary in size corresponding to the ⟨*clrfcn*⟩. The result will be disappointing unless there is considerable contrast between the lightest and darkest grays of the gradient. Therefore, it is recommended that the color function cover the entire range from 0 to 1 (black to white). There are no particular memory problems with gradients in METAFONT, at least no more so than gray fills.

4.5.3   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 penstrokes and fills (with possible clipping).

\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 or indeed any TEX commands, by using the \setrender command. This redefinition is local when it occurs inside an `mfpic` environment, so there it can be enclosed in braces to restrict its range. Outside an `mfpic` environment it is a global redefinition.

For example, suppose one uses \setrender{\dashed\shade} in a `mfpic` environment. If the command \circle{(0,0),1} occurs later in that environment, it will produce a shaded circle with a dashed outline. If an explicit rendering prefix is given in a drawing command, it will override this default.

4.5.4   Examples

It may be instructive, for the purpose of understanding the syntax of *shape-modifier and rendering prefixes*, to consider two examples:

        \draw\gfill[red]\lclosed\polyline{...}

which fills inside a polygon and draws its outline; and

        \gfill[red]\lclosed\draw\polyline{...}

which draws all of the outline *except* the line segment supplied by \lclosed, then fills the interior. Thus, in the first case the path is first defined (by \polyline), then closed, then the resulting closed path is filled, and finally drawn. In the second case the order is: defined, drawn, closed, filled. In particular, what is drawn in the second case is the path not yet closed. It should also be pointed out that in the last case, the fill is placed last and will cover half the thickness of the previously drawn outline.

4.6   Functions and Plotting.

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

4.6.1   Defining functions

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

Defines a METAFONT function ⟨*fcn*⟩ of the parameters ⟨*param1*⟩, ⟨*param2*⟩, ..., by the META-FONT 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 who know what this means, the name should have no suffixes.) Try to make the name 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 META-FONT statements, followed by an expression, all enclosed between begingroup and endgroup. The \fdef command automatically supplies these grouping commands around the definition so if the entire ⟨*mf-expr*⟩ is one such compound expression the user need not type them. METAFONT functions can call METAFONT functions, even recursively.

Many common functions have been predefined in grafbase, which is a package of METAFONT macros that implement MFPIC's drawing. These include the rest of the 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, coth, sech, and csch; and some of their inverses: ln (or log), asinh, acosh, and atanh are also defined.

There are also two conversion functions: radians(t) produces the number of radians in t degrees and degrees(t) produces the number of degrees in t radians. In these expressions the special variable pi produces π, accurate to roughly 5 decimals. (METAFONT and METAPOST provide accuracy only to $\pm 2^{-17} = \pm .76 \times 10^{-5}$.)

The integer functions gcd(m,n) and lcm(m,n) produce the greatest common divisor and least common multiple of two integers m and n.

4.6.2 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 4.2.5) 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*⟩)/⟨*step*⟩ is not a whole number, the nearest whole number of 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. All the macros are figure macros, defining a path to which prefixes may be applied.

`\function[`⟨*spec*⟩`]{`⟨$x_{\min}$⟩`,`⟨$x_{\max}$⟩`,`⟨Δx⟩`}{`$f(\mathtt{x})$`}`

This figure macro produces an approximation to the graph of $y = f(x)$, where f is 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⟩`}{(`$x(\mathtt{t}),y(\mathtt{t})$`)}`
`\parafcn[`⟨*spec*⟩`]{`⟨$t_{\min}$⟩`,`⟨$t_{\max}$⟩`,`⟨Δt⟩`}{`⟨*pair-fcn*⟩`}`

This figure macro produces the parametric path determined by the last argument. This can be a pair of expressions $x(\mathtt{t})$ and $y(\mathtt{t})$ enclosed in parentheses and separated by a comma, with the literal variable t. Alternatively, the last argument can be a METAFONT function or expression in t that returns a pair.[16] 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[`⟨*spec*⟩`]{`⟨$\theta_{\min}$⟩`,`⟨$\theta_{\max}$⟩`,`⟨$\Delta\theta$⟩`}{`$f(\mathtt{t})$`}`

This figure macro produces the graph of the polar coordinate equation $r = f(\theta)$, where f is a METAFONT numeric function or expression of one numeric argument, and θ varies from ⟨$\theta_{\min}$⟩ to ⟨$\theta_{\max}$⟩ in steps of ⟨$\Delta\theta$⟩. Each θ value is interpreted as an angle measured in *degrees*. In the expression $f(\mathtt{t})$, the unknown t stands for θ. The default ⟨*spec*⟩ is s. For example

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

draws one loop of a 4-petal rosette. Note that this function demands the variable t be in degrees. The range and step size must be in degrees and the function must operate on the numeric variable t in degrees. If one needs to measure angles in radians, use the conversion functions degrees() and radians(), as follows:

```
\plrfcn{0,degrees(pi/2),degrees(pi/36)}{sin (radians(2t))}
```

`\btwnfcn[`⟨*spec*⟩`]{`⟨$x_{\min}$⟩`,`⟨$x_{\max}$⟩`,`⟨Δx⟩`}{`$f(\mathtt{x})$`}{`$g(\mathtt{x})$`}`
`\btwnplrfcn[`⟨*spec*⟩`]{`⟨$\theta_{\min}$⟩`,`⟨$\theta_{\max}$⟩`,`⟨$\Delta\theta$⟩`}{`$f(\mathtt{t})$`}{`$g(\mathtt{t})$`}`

These are figure macros. The first one produces a closed path surrounding the region between the graphs of the two functions. The second one does the same for two polar functions. That is (in both cases), the path follows the first function (in order or increasing x or θ), thence along the straight line to the *end* of the second one, thence backwards along the second function (decreasing x or θ) and finally along the straight line to the start. The last two mandatory arguments, the functions, are specified exactly as in `\function` and `\plrfcn`, being numeric functions of one numeric argument x or t. Unlike the previous function macros, the default ⟨*spec*⟩ is p—these macros are intended to be used for shading between drawn functions, a task for which smoothness is usually unnecessary. For example, the first line below

---

[16]There are very few of these. METAFONT provides dir t, which is essentially (cosd t, sind t). MFPIC adds cis t which is (cos t, sin t).

```
\shade\btwnfcn{0,1,.1}{0}{x - x**2}
\btwnplrfcn[s]{-30,30,5}{1}{2*cosd 2t}
```

shades the area between the *x*-axis and the given parabola. The second draws the boundary of the region between the circle $r = 1$ and one loop of the rosette $r = 2\cos 2\theta$.

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

```
\lclosed\connect
\function{⟨xmin⟩,⟨xmax⟩,⟨Δx⟩}{f(x)}
\reverse\function{⟨xmin⟩,⟨xmax⟩,⟨Δx⟩}{g(x)}
\endconnect
```

`\lclosed` was described in subsection 4.4.1 and the `\connect`…`\endconnect` pair was described in subsection 4.4.2.

`\belowfcn[⟨spec⟩]{⟨xmin⟩,⟨xmax⟩,⟨Δx⟩}{f(x)}`
`\plrregion[⟨spec⟩]{⟨θmin⟩,⟨θmax⟩,⟨Δθ⟩}{f(t)}`

These figure macros produce identical results to `\btwnfcn` and `\btwnplrfcn` when the first function is just 0. They are, however, much more efficient. The first of these, `\belowfcn`, produces the path surrounding the region bounded by the *x*-axis, the graph of $y = f(x)$ and the two vertical lines $x = x_{\min}$ and $x = x_{\max}$. (The region is not actually *below* $y = f(x)$ unless $f(x) \geq 0$ throughout the interval.) The second produces the path surrounding the region bounded by the polar function $r = f(\theta)$ and the two rays $\theta = \theta_{\min}$ and $\theta = \theta_{\max}$.

The arguments of these command are the same as the nonclosed versions, `\function` and `\plrfcn`, except the default for the optional agument is `[p]`. Again, this is because it is mainly for shading. However, drawing the boundary is often needed:

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

shades one loop of the 4-petal rosette, and then draws it.

The next sets of macros are similar to the previous function plotting macros, but don't fit the ⟨*max*⟩, ⟨*min*⟩ ⟨*step*⟩ model for the first argument. For the first (`\levelcurve`) this is a limitation of the task being performed. For the others (`\DEgraph`, `\DEtrajectory`) it is a design choice.

`\levelcurve[⟨spec⟩]{⟨seed⟩,⟨step⟩} {⟨inequality⟩}`

This figure macro produces a level curve of some function $F(x,y)$. There are three requirements on the parameters for this to work correctly. First, in order to obtain the curve satisfying $F(x,y) = C$, the {⟨*inequality*⟩} must be either {`F(x,y) > C`} or {`F(x,y) < C`}.[17] Second, the level curve must surround the point given by the ⟨*seed*⟩ paramter, and third, the inequality must be true at this seed point.

The command works by searching rightward from ⟨*seed*⟩ until it encounters the first point on the level curve. It then tries to find a nearby point on the level curve and joins it to the first one, and continues similarly until it finds it has returned near the starting point. The meaning of "nearby point on the level curve" is the intersection of the level curve with a circle of radius ⟨*step*⟩ centered at the previously found point. If the region defined by the inequality extends beyond the bounds of the picture (as set by the `\mfpic` command), the region is truncated and the resulting curve will follow along the picture's border.

Since the algorithm only approximates the level set, a tolerance (how close the points are to actually being *on* the level curve) is chosen which gives two decimal places more accuracy than

---

[17] A non-strict inequality such as `>=` can be used, but the result will not be significantly different.

$\langle step\rangle$. The value of $\langle step\rangle$ is interpreted in *graph* units and so should be a pure number. The [$\langle spec\rangle$] is either [p], in which case the calculated points are joined with straight lines, or [s$\langle tension\rangle$] as in \function. The default is [s]: a smooth curve with the current default tension.

In general, choosing a $\langle step\rangle$ that corresponds to a few millimeters works reasonably well. For example, if the graph unit is 1cm (for example, \mfpicunit=1cm and no scaling is used), then $\langle step\rangle = 0.5$ might be a reasonable first choice. If the level set is reasonably smooth and [s] is used, then the result will match the actual curve to within .005cm, which is approximately .14pt, which is less than half the thickness of the standard pen used to draw it.

Be warned that there is a limit: there should not be more than 2000 steps in the completed curve. In a figure which is 10-by-10 graph units, a level curve without too much oscillation would probably be less than 80 units in length and a step size of .04 would probably produce under 2000 steps. This should be accurate enough for most purposes. If you *really* need more, the value of the METAFONT variable max_points must be changed. This can be done with \setmfvariable (see section 4.12.4).

As a special case, if $\langle step\rangle$ is 0, the maximum of width and height of the figure (as given by the arguments to the mfpic environment) is divided by 100. For example, in a 5-by-10 graph, giving a step size of 0 will actually select $\langle step\rangle = 10/100 = 0.1$.

The algorithm used will produce imprecise results if there are two points on the curve closer than $\langle step\rangle$ in straight-line distance, but much further apart when measured along the curve.

\DEgraph[$\langle spec\rangle$]{$\langle x_0\rangle$,$\langle y_0\rangle$,$\langle \Delta s\rangle$,$\langle N\rangle$}{f(x,y)}
\DEtrajectory[$\langle spec\rangle$]{$\langle p_0\rangle$,$\langle \Delta s\rangle$,$\langle N\rangle$}{$\mathbf{F}$(x,y,t)}

The first of these plots the graph of the solution of the differential equation

$$\frac{dy}{dx} = f(x,y), \quad y(x_0) = y_0.$$

The $\langle \Delta s\rangle$ parameter is a step size and the $\langle N\rangle$ parameter is the number of steps. The step size is *not* an increment in the *x* variable. Rather is is (roughly) the distance from one point to the next along the graph as METAFONT computes them. That is, METAFONT computes using a variable *x*-step Δx, chosen so that $\sqrt{\Delta x^2 + \Delta y^2}$ is approximately $\langle \Delta s\rangle$. The algorithm used is a modified 4-step Ringe-Kutta method.

The second macro, \DEtrajectory draws the path traced by the solution $(x(t), y(t))$ of

$$\left(\frac{dx}{dt}, \frac{dy}{dt}\right) = \mathbf{F}(x,y,t), \quad (x(0), y(0)) = p_0.$$

This is not a *graph*, since the dependence on *t* cannot be shown in two dimensions (a third dimension would be needed). The parameter $\langle p_0\rangle$ should be an ordered pair of numbers, the $\langle \Delta s\rangle$ and $\langle N\rangle$ are as for \DEgraph. The function $\mathbf{F}(x,y,t)$ should be either a pair-valued expression or an ordered pair of numeric expressions. The variables must be literally x, y and t. The expressions do not have to explicitly depend on these variables. In fact, the \DEgraph macro is implemented using the same internal macro as \DEtrajectory with $\mathbf{F}(x,y,t) = (1, f(x,y))$ and $p_0 = (x_0, y_0)$.

Notice that the trajectory starts at $t = 0$. If you need some other starting value $t = a$, then replace *t* in the formula for $\mathbf{F}(x,y,t)$ with $(t + a)$.

It is possible to use a negative value of Δs in both these macros. For \DEgraph this produces the graph to the left of $x = x_0$, and for \DEtrajectory it produces the trajectory with time running backward. For the latter, it is also equivalent to replacing $\mathbf{F}(x,y,t)$ by its negative.

The purpose of making $\langle \Delta s\rangle$ a distance rather than an *x*-increment or a *t*-increment (as the Runge-Kutta method is taught in the usual mathematics courses) is stability: even very simple differential

equations can have graphs the tend to ∞ in finite time. These macros, however, never travel more than a distance $N\Delta s$ from the starting point.

If you want to use MFPIC to illustrate the results of the standard Runge-Kutta method or other methods, you can use the MFPIC4ODE package. That package also includes the Euler method and the two-step Runge-Kutta method. It loads MFPIC if it has not already been loaded. Like MFPIC, it works in plain TEX (with `\input mfpic4ode`) or LATEX (with `\usepackage{mfpic4ode}`).

4.6.3 Plotting external data files

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

The figure macro `\datafile` produces a curve connecting the points listed in the file ⟨file⟩. (The context makes it clear whether this meaning of `\datafile` or that of subsection 4.2.2 is meant.) The ⟨spec⟩ 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 ⟨spec⟩ 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 4.2.2 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 (omitting aliases) of MFPIC macros that allow this usage of `\datafile`:

– Numeric data: `\barchart`, `\dashpattern`, `\numericarray`, `\piechart`, and all the axis marks commands.
– Point or vector data: `\cbeziers`, `\closedcbeziers`, `\closedcomputedspline`, `\closedcspline`, `\closedmfbezier`, `\closedqbeziers`, `\closedqspline`, `\computedspline`, `\convexcurve`, `\convexcyclic`, `\cspline`, `\curve`, `\cyclic`, `\fcncurve`, `\fcnspline`, `\mfbezier`, `\periodicfcnspline`, `\plotsymbol`, `\point`, `\polygon`, `\polyline`, `\putmfpimage`, `\qbeziers`, `\qspline`, `\turtle`, and `\pairarray`.

In addition `\setarray` and `\globalsetarray` (with the numeric or pair data type) allow this usage.

`\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 (numbers separated by spaces, with the first two determining the (x, y) pair) corresponds to the following 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 described default for pair data. The command `\usingnumericdefault` is the equivalent of `\using{#1 #2}{#1}`, a useful default for numeric data.

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 allows one to overcome this.

`\makepercentother`
`\makepercentcomment`

Here is an example or their use:

```
\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. Moreover, the above construction will fail inside the argument of another command.

`\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` causes `\plotdata` to use the rendering command `\draw` with a color option that cycles through eight different colors starting with black (hey! black is a color too). The command `\pointedlines` causes `\plotdata` to use the rendering command `\plot`, cycling through nine symbols. The command `\datapointsonly` causes `\plotdata` to use the rendering command `\plotnodes`, cycling through the same nine symbols. The data points become the nodes of the paths created and so only the data points are plotted. The command `\dashedlines` restores the default. See appendix 5.4 for the details on the actual dash patterns, colors and symbols used.

The command `\coloredlines` will produce a warning under the metafont option and substitute `\dashedlines`. Under the metapost option, this is the sole exception to the general rule that all curves are drawn in `drawcolor` by default: the `\plotdata` command after `\coloredlines` has been issued.

Note that MFPIC always creates a path internally. It is possible that your data is not path-like and what you want is a scatter-plot. Simply use `\datapointsonly` and the effect is the same: METAPOST builds a polygonal path connecting all the points in your file, but when it plots the path, it only places a dot (or other symbol) at each data point.

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 appendix 5.4 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 dashes, 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 be changed: 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 color names defined using \mfpdefinecolor. The symbols can be any of those listed with the \plotsymbol command (see subsection 4.2.1), or any known METAFONT path variable. The colors can also be METAPOST color constants or expressions, and the symbols can be expressions of type path. In recent METAPOST these 'colors' can be numeric (selecting gray), rgbcolor or cmykcolor. Within a mfpic environment, the changes made are local to that environment. Outside, they affect all subsequent environments.

Using \reconfigureplot{colors} under the metafont option will have no effect, but may produce an error from METAFONT unless the colors used conform to the guidelines in subsection 4.3.4. This also holds for \defaultplot{colors} (below).

\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.

4.7 Labels and Captions.

4.7.1 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⟩ ...}

These place TeX text or math on the graph. 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, { and }. If braces are used, any number of coordinate pairs may be listed, separated by commas. This is what is meant by ⟨*pair-list*⟩ in the above syntax. If mplabels is in effect, the ⟨*pair-list*⟩ can be any list of expressions recognized as a pair by METAPOST.

The optional parameter [⟨*just*⟩] specifies the *justification*, the relative placement of the label with respect to the point with coordinates (⟨*x*⟩,⟨*y*⟩). It is a two-character sequence in which 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]: the left end of the baseline of the text is placed at the coordinates.

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 \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.

There is another command, \startbacktext, which also save the labels and adds them later, but *under* the rest of the picture as background text. Thus, they will not be clipped, but may be covered up. Since erasing regions with \gclear actually covers up those regions with white, labels saved as background text may appear to have portions erased.

\everytlabel{⟨*TEX-code*⟩}

One problem with multiline \tlabels 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 each line of a

tlabel is wrapped in a box, but the commands of \everytlabel are outside all of them, so no actual text should be produced by the contents of \everytlabel.

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. If mplabels is not turned on later, that rotation will be ignored by \tlabel.

\tlabeloffset{⟨*hlen*⟩}{⟨*vlen*⟩}
\tlpointsep{⟨*len*⟩}
\tlpathsep{⟨*len*⟩}
\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 \tlpointsep 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.

Prior to version 0.8, this separation also defined the separation between the label and those curves designed to frame the label such as \tlabelrect (subsection 4.7.2). Now the two separations are independent and \tlpathsep is used to set the separation between the label and such paths.

For backward compatability, the command \tlabelsep is still available and sets both separations to the same value.

\axislabels{⟨*axis*⟩}[⟨*just*⟩]{{⟨*text₁*⟩}⟨*n₁*⟩,{⟨*text₂*⟩}⟨*n₂*⟩,...}

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 \tlpointsep 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, \plottext places a copy of ⟨*text*⟩ at each of the listed points. Since MFPIC version 0.9, when \tlabel was enhanced to allow lists of points, it

is implemented by an equivalent \tlabel command and is only kept for backward compatibility. It differs from \tlabel when the optional argument is absent: the default justification is [cc] regardless of the setting of \tlabeljustify.

\mfpverbtex{⟨*T_EX-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 ⟨*T_EX-cmds*⟩ in the argument are written to the .mp file, preceded by the METAPOST command verbatimtex and followed by etex. Line breaks within the ⟨*T_EX-cmd*⟩ are preserved. There is also a linebreak between the end of ⟨*T_EX-cmds*⟩ and the etex. 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 is issued after \opengraphsfile, its contents are immediately written to the .mp file. If it is issued before \opengraphsfile, its contents are saved and written when the file is opened (successive uses being cummulative). In this case its contents will precede the boilerplate T_EX code that MFPIC writes. If you wish to redefine some of that code, you need to use \mfpverbtex after \opengraphsfile.

Because of the way METAPOST handles verbatimtex material, the effects cannot be constrained by any grouping unless one places T_EX grouping commands within ⟨*T_EX-cmds*⟩. However, MFPIC itself places grouping commands into the output file at the beginning and end of each picture, so definitions written by a \mfpverbtex are local to any picture in which it occurs. Prior to version 0.8, MFPIC did not write comments that occured within the ⟨*T_EX-cmds*⟩. Now they will be preserved, and can be used to place the '%&latex' line that some T_EX distributions permit as a signal that latex should be run to produce the labels.

This command attempts a near-verbatim writing of the ⟨*T_EX-cmds*⟩ and, as with all verbatim-like commands, it should not be used in the argument of another command.

\startbacktext ... \stopbacktext

When T_EX adds labels (\nomplabels) they have to be positioned either on top of a complete figure, or placed under a complete figure. The most reasonable choice (and happily the easiest to implement) is to put them on top. When METAPOST is placing labels (option mplabel) the same can be forced with the option overlaylabels, but otherwise they are placed as they occur, with later drawing commands perhaps putting their results on top of the labels or clipping parts of them off.

Sometimes it is useful to place some label as a background (not on top), and yet not have it clipped by later commands. The effect of the command \startbacktext is that \tlabel commands are saved in a special place until the command \stopbacktext. Then, at \endmfpic the rest of the figure is simply place on top of them. Since labels in METAPOST files can only consist of characters from some font, if one wants to include a graphic in the background (for example, via \includegraphics), one needs to switch off mplabels:

```
\nomplabels
\startbacktext
  \tlabel[cc](0,0){\includegraphics{mygraph}}
\stopbacktext
\usemplabels
```

As with other labels, it is permitted to switch mplabels off and on while creating background text. If there are both kinds of labels within the background text area the ones handled by T_EX will be further back than the ones handled by METAPOST. Within a given type, earlier ones are further back than later ones.

MFPIC normally uses a naming scheme like \cmd . . . \endcmd and tries to arrange that cmd can be used as an environment. As currently written, the extra grouping added by \begin{cmd} and \end{cmd} would break the code that implements background text, so we have named these in a different way to avoid suggesting this possiblity. There should be at most one of these pairs in any mfpic environment. It can occur anywhere in the environment, but the two commands must not be inside any grouping.

Under the metapost option, the \gclear command doesn't really clear a space, but rather paints the space over with white. Any background text will not be visible through such 'holes'. This is a limitation of METAPOST.

\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 may typeset its argument twice (as might LATEX's \caption), the first time as a single line to test its width, then again if that was too wide. 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 default is to set lines both left- and right-justified (except for the last line) with no indentation on the first line. If the option raggedcaptions is in effect, the lines are only left-justified and ragged on the right. Finally, if the option centeredcaptions is in effect, each line of the caption will be centered (under raggedcaptions they will be ragged on both sides).

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

4.7.2   Curves surrounding text

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

This figure macro 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 (the optional justification argument for the \tlabel command) 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 argument 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 \tlpathsep. All other versions produce the smallest described curve that contains this rectangle.

These commands may be preceded by prefix macros (see the sections 4.4 and 4.5, above). They all have a 'star-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 figure macro is similar to \tlabelrect, except it produces 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*...

This figure macro produces 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 takes 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 4.7.1.

4.8 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}. In a LaTeX docu-

ment, \newsavepic is actually defined to be \newsavebox.

\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. The \usepic command is essentially a clone of the LATEX \usebox command. Since the contents of the saved picture are only defined during the TEX run, \usebox cannot be used in the TEX-commands argument of the \tlabel command while mplabels is in effect.

4.9   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 number (and any text placed by \tlabel and its relatives without mplabels in effect). 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 commands 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 environment syntax: \begin{mfpframe} ... \end{mfpframe}.

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 or command of the user's choice or devising.

4.10   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 nor \tcaption) MFPIC provides the ability to apply METAFONT affine transforms.

4.10.1   TRANSFORMING 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. In LATEX, one can use the environment named coords.

Transforms provided by MFPIC:

`\rotate{`⟨θ⟩`}`	Rotate around origin by ⟨θ⟩ degrees.
`\rotatearound{`⟨p⟩`}{`⟨θ⟩`}`	Rotate around point ⟨p⟩ by ⟨θ⟩ degrees.
`\turn[`⟨p⟩`]{`⟨θ⟩`}`	Rotate around point ⟨p⟩ (origin is default) by ⟨θ⟩.
`\reflectabout{`⟨p₁⟩`}{`⟨p₁⟩`}`	Reflect in the line through points ⟨p_1⟩ and ⟨p_2⟩.
`\mirror{`⟨p₁⟩`}{`⟨p₂⟩`}`	Same as `\reflectabout`.
`\shift{`⟨v⟩`}`	Shift origin by the vector ⟨v⟩.
`\scale{`⟨s⟩`}`	Scale uniformly by a factor of ⟨s⟩.
`\xscale{`⟨s⟩`}`	Scale only the *x* coordinates by a factor of ⟨s⟩.
`\yscale{`⟨s⟩`}`	Scale only the *y* coordinates by a factor of ⟨s⟩.
`\zscale{`⟨pair⟩`}`	Scale by the length of vector ⟨v⟩, and rotate by its angle.
`\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 `grafbase.dtx`.
`\boost{`⟨χ⟩`}`	Special relativity boost by χ, see `boost` in `grafbase.dtx`.
`\xyswap`	Exchange the values of *x* and *y*.
`\applyT{`⟨transformer⟩`}`	Apply the ⟨transformer⟩.

`\applyT` is for METAFONT hackers. Any code is permitted that satisfies METAFONT's syntax for a ⟨transformer⟩ (see D. E. Knuth, "The METAFONTbook", page 73), 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: STz usually means to apply S to the result of Tz).

Finally, some of these may not produce what the unwary user might expect if the `mfpic` environment was started with unequal scaling. For example, in such a case a rotated rectangle will not have right angles unless the rotation is by a multiple of 90 degrees. The reason for this: the scaling given by the `\mfpic` command is applied last and slanted lines subjected to unequal horizontal and verical scaling will change have their angles changed.

4.10.2 TRANSFORMING 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 and 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 allows the following path to be an argument, that is, enclosed in braces. This is solely to support files written for past MFPIC versions in which \store was *not* defined as a prefix macro.

One use of \store is to create a shorthand for a path that is otherwise long and tedious to type. Another is to create 'symbols' or 'arrowheads' for use in \plotsymbol, \arrowhead and related commands.

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

This figure macro produces the path represented by ⟨*path expression*⟩, which is either a path variable in which a path was previously stored, or a valid METAFONT expression combining such variables and constant paths. This allows the use of path variables or expressions as figure macros, permitting all prefix operations, etc.. Here are some examples of the use of \store and \mfobj.

```
\store{my_f}{\cyclic{...}}                 % Store a closed curve.
\dotted\mfobj{my_f}                        % Now draw it dotted,
\hatch\mfobj{my_f}                         % and hatch its interior
% Create two symbols
%   one outline:
\store{MyTriang}{\polyline{(-.5,-.5),(.5,-.5),(0,.5),(-.5,-.5)}
%   one solid:
\store{MySolidTriang}\polygon{(-.5,-.5),(.5,-.5),(0,.5)}
% Use them as symbols:
\plotsymbols{MyTriang}{(0,0),(2,2)}
\arrowmid{MySolidTriang}\polyline{(1,1),(0,2)}
```

Note: If a stored path has the same starting point as ending point, but is *not* closed then it will behave like Circle (for example) when used in \plotsymbol: only its outline is drawn, and its interior is erased when clearsymbols is in effect. If a closed path is stored, it behaves like SolidCircle: it is not drawn, but rather filled. If a path is stored that satisfies neither, it behaves like Asterisk, being simply drawn in all circumstances.

The two forms \mfobj and \mpobj are absolutely equivalent; they differ only in spelling.

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 recently completed figure macro (possibly modified by prefixes).

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, the rendering macros have the side effect of adding ink to the page (or subtracting it). To know where to place this ink, a calculation is performed that translates graph coordinates to actual positions. The above transforms work by modify the parameters used in that calculation. On the other hand, \store merely stores the output of the immediately following prefix or figure macro. See the beginning of section 4.4 for a discussion of input, output and side effects of MFPIC prefix and figure macros.

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.

\rotatepath{$\langle p \rangle$,$\langle \theta \rangle$}...
\shiftpath{$\langle v \rangle$}...
\scalepath{$\langle p \rangle$,$\langle s \rangle$}...
\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{$\langle transformer \rangle$}...

These are modifying macros that all return the result of applying an affine transformation to the following path. They differ in the transformation applied and the data needed in the mandatory argument. I have found them extremely useful, and better than coords environments when I need to draw a figure, together with several slightly different versions of it. If \store is used just before one of these prefixes, it stores the transformed path rather than the original.

\rotatepath rotates the following path by $\langle \theta \rangle$ degrees about point $\langle p \rangle$.

\shiftpath shifts the following path by the vector $\langle v \rangle$.

\scalepath scales (magnifies or shrinks) the following path by the factor $\langle s \rangle$, in such a way that the point $\langle p \rangle$ 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. Points below that line move left. 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. However, after a swap, verticals become horizontal and horizontals become vertical. (It is impossible, when the scales are different, for an affine transform to both preserve shape and exchange horizontal and vertical lines.)

This compensation for different scales is also done for \rotatepath, so the resulting path always has the same shape after the rotation as before. None of the other path transformation prefixes compensate for different scales, and none of the coordinate system transformations of the previous subsection do it.

For METAFONT or METAPOST power users, \transformpath can take any 'transformer' and transform the following path with it. Here, a *transformer* is the same as in the previous section. Examples are scaled, shifted(1,1), and rotatedabout(0,1). Note that using this last transformer with \transformpath is almost like \rotatepath{(0,1)}, but it does not compensate for different scales.

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.

One complication is the handling of the default rendering. One expects

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

to draw a rectangle, and

    \rotatepath{(0,0),45}\rect{(0,0),(1,1)}

to draw a rotated rectangle (but not the original). That is, a transformation + figure is treated as if it were a single figure. But what would one expect in the following?

    \rotatepath{(0,0),45}\dashed\rect{(0,0),(1,1)}

What one will get is the original dashed and the rotated one with the default rendering (typically drawn with solid lines). That is, these prefixes cannot see the renderings that occur later in the sequence. They add the default rendering as if those didn't exist. If something other than this is desired, one can either rearrange the prefixes or add a \norender in appropriate places. For example, to add a shifted arrowhead without drawing the shifted path:

    \arrow\norender\shiftpath{(0,1)}\arrow\draw\lines{(0,0),(8,8)}

4.11   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 only to mfpic environments if set inside one, 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 them consistent.

There are a few parameters that do obey TEX grouping, but only inside mfpic environments. These are noted where the parameter is described.

All parameters are initialized when MFPIC is loaded. We give the initial value or state in each of these descriptions.

\mfpicunit

This dimension stores the basic unit length for MFPIC pictures. The *x* and *y* scales in the \mfpic macro are multiples of this unit. The initial value is 1pt. It is global outside an mfpic environment. Changes made to it inside an mfpic environment have no effect and are lost at the end of the environment.

\pointsize

This dimension stores the diameter of the circle drawn by the \point macro and the diameter of the symbols drawn by \plot, \plotsymbol and \plotnodes. The initial value is 2pt.

\pointfilltrue, \pointfillfalse

This TEX boolean switch determines whether the circle drawn by \point will be filled or open (outline drawn, inside erased). The initial state is true: filled. This value is local to any TEX group inside an mfpic environment. Outside such it is global.

\pen{⟨*size*⟩}
\drawpen{⟨*size*⟩}
\penwd{⟨*size*⟩}

These commands establishes the width of the normal drawing pen (that is, the thickness of lines, whether solid or dashed). The initial value is 0.5bp. This width is stored by METAFONT. This has no effect on the size of dots for \dotted, \shade, \grid, etc. It also has no effect on the lines drawn for hatching. 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{⟨*diam*⟩}

This command sets the diameter of the dots used in the shading macro. The drawing and hatching pens are unaffected by this. The initial value is 0.5bp, and the value is stored by METAFONT.

\hatchwd{⟨*size*⟩}

This sets the line thickness used in the hatching macros. The drawing pen and shading dots are unaffected by this. The initial value is 0.5bp, and the value is stored by METAFONT.

\polkadotwd{⟨*diam*⟩}

This sets the diameter of the dots used in the \polkadot macro. The initial value is 5bp, and the value is stored by METAFONT.

\headlen

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

`\axisheadlen`

This dimension stores the length of the arrowhead drawn by the `\axes`, `\xaxis` and `\yaxis` macros, and by the macros `\axis` and `\doaxes` when applied to the parameters x and y. The initial value is 5pt.

`\sideheadlen`

This dimension stores the length of the arrowhead drawn by the `\axis` and `\doaxes` macros when applied to l, b, r or t. The initial value is 0pt (that is, the default is not to put arrowheads on border axes).

`\headshape{`⟨*ratio*⟩`}{`⟨*tension*⟩`}{`⟨*filled*⟩`}`

This establishes the shape of the `Arrowhead` drawn by the `\arrow...` and `\axes` macros. It also establishes the shape of `Leftharpoon` and `Rightharpoon`. The value of ⟨*ratio*⟩ is the ratio of the width of the arrowhead to its length; ⟨*tension*⟩ is the tension of the Bézier curves; and ⟨*filled*⟩ is a METAFONT boolean value indicating whether the arrowheads are to be filled (if true) or open. The initial values are 1, 1, and false, respectively. Setting ⟨*tension*⟩ to the literal keyword 'infinity' will make the sides of the arrowheads straight lines. The harpoon heads are arranged to be exactly half of the full arrowhead. The ⟨*ratio*⟩, ⟨*tension*⟩ and ⟨*filled*⟩ values are stored by METAFONT.

After `\headshape` is used, the symbols `Arrowhead`, `Leftharpoon`, and `Rightharpoon` take on the new shape if used in one of the `\plot...` commands.

`\dashlen, \dashspace`

These 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 $1/n$ 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 initial 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 shorthands for certain settings of the `\dashlen` and `\dashspace` dimensions. The macro `\dashlineset` sets both values to 4pt, while `\dotlineset` sets `\dashlen` to 1pt and `\dashspace` to 2pt. They are kept mainly for backward compatibility.

`\hashlen`

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

`\shadespace`

This dimension establishes the spacing between dots drawn by the `\shade` macro. The initial 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. Under METAFONT it is possible that using one of these macros can have no visible effect. See the discussion of the `\shade` macro in subsection 4.5.2.

\polkadotspace

This dimension establishes the spacing between the centers of the dots used for the macro \polkadot. The initial value is 10pt.

\dotsize, \dotspace

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

\griddotsize

This dimension gives the default sizes of dots in the \grid and \plrgridpoints commands. The initial value is 0.5pt

\symbolspace

Similar to \dotspace, this TEX dimension establishes the space between the centers of symbols placed by the macro \plot{⟨*symbol*⟩}.... Its initial value is 5pt.

\hatchspace

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

\tlpointsep{⟨*separation*⟩}
\tlpathsep{⟨*separation*⟩}
\tlabelsep{⟨*separation*⟩}

The first 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. The second sets the separation between the text and the curve defined by the commands \tlabelrect, \tlabeloval or \tlabelellipse. The third sets both of these separations to the same value. It is for backward compatibility: in the past there was only one dimension used for both purposes. The initial value of each is 0pt. The values are 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 initial state is to have both horizontal and vertical offsets of 0pt. The values are stored by both TEX and METAFONT.

\mfpdataperline

When MFPIC is reading from data files and writing to the output file, this macro stores the maximum number of data points that will be written on a single line in the output file. Its initial definition is \def\mfpdataperline{5}. Any such definition (or redefinition) obeys *all* TEX groupings.

\mfpicheight, \mfpicwidth

These 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. As they are not to be changed, grouping is irrelevent, but when MFPIC sets them, it does so globally. These are set even if the picture is saved with \savepic. If they are needed for the corresponding \usepic, and that occurs after another mfpic environment, they should be copied to other length commands right after the mfpic environment that set them.

`\mfpiccaptionskip`

This skip register ('rubber length' in LaTeX) stores the space between a picture and the caption produced with `\tcaption`. It is local to all TeX groups. If changed inside an `mfpic` environment it will affect only the `\tcaption` command in that picture. It's initial setting is `\medskipamount`, producing the same space as a `\medskip`.

4.12 For Advanced Users.

4.12.1 SPLINES

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

These figure macros use alternative ways of defining curves. In each case, ⟨*list*⟩ is a comma separated list of ordered pairs. 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.

For `\qspline`, the curve will pass through the midpoints of the line segments joining the points in the list, tangent to that line segment.

For the `\cspline`, the list also defines line segments. Divide these into equal thirds at two points on each segment. Connect these *division points only* to obtain line segments. Each *odd numbered* segment is the middle third of one of the original line segments. The `\cspline` curve passes through the midpoint of each *even numbered* line segment, tangent to it.

`\computedspline{⟨`*list*`⟩}`
`\closedcomputedspline{⟨`*list*`⟩}`

These figure macros both produce cubic splines. For these you *do* provide the list of points the curves are to pass through. They become the nodes, and then the control points are computed from them. The nodes do not uniquely determine the control points so extra equations are required. For the first version, the extra equations give the path zero curvature at the endpoints (a *relaxed* spline). For the closed version, the extra equations are those that close the curve smoothly. The portions of the spline that connect one node to the next are parametrized cubic Béziers, they are computed so that the first and second derivatives (with respect to the parameter) of adjacent curves match at the common node.

`\fcnspline{⟨`*list*`⟩}`
`\periodicfcnspline{⟨`*list*`⟩}`

These figure macros use cubic spline equations (as in `\computedspline` above) to produce a smooth graph of a function based on a list of points with increasing *x*-values. See `\fcncurve` in section 4.2.5 for another way to do this. As in the computed splines, above, the spline equations at the nodes do not provide sufficient information to compute all control points. In the basic version, `\fcnspline`, extra equations produce a graph with zero curvature at the endpoints (a relaxed spline), while the periodic version uses equations that make the first and second derivatives at the last point match those at the first point.

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

These are modifying macros that close the following path. 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.

4.12.2 BÉZIERS

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*⟩`}`

These figure macros uses the METAFONT path join operator '`..tension `⟨*tens*⟩`..`' to connect the points in the list. 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{`⟨*list*⟩`}`
`\closedqbeziers{`⟨*list*⟩`}`

These figure macros produce *quadratic* Bézier curves, the equivalent of a sequence of LATEX `\qbezier` commands. Note the plural forms, to distinguish the first from the LATEX command, and to indicate that they can 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. If the number of ponts is wrong, no error is produced: the last point is simply repeated to get the required number.

The curve will not automatically be smooth; that depends on the choice of the control points.

`\cbeziers{`⟨*list*⟩`}`
`\closedcbeziers{`⟨*list*⟩`}`

These figure macros produce a series of *cubic* Bézier curves. In the ⟨*list*⟩, the first, fourth, seventh, etc., are the points to connect, while the second and third, fifth and sixth, etc., are pairs of control points. The closed version uses the starting point as the ending point, and so needs a number of points divisible by 3 ($n = 3k$). The open version requires an explicitly given ending node (so $n = 3k + 1$). If the number of ponts is wrong, no error is produced: the last point or last two points are simply repeated to get the required number.

The curves will not automatically be smooth; that depends on the choice of the control points. Cubic Béziers are how curves are represented in PostScript files, and how a number of vector drawing programs represent curves.

4.12.3 RAW METAFONT CODE

\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.[18] 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 semi-colon (';') 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 extensive example. It should produce the word 'mfpic' in blue, outlined in green in a box with yellow background.

4.12.4 CREATING METAFONT VARIABLES

\setmfvariable{⟨*type*⟩}{⟨*name*⟩}{⟨*value*⟩}
\setmpvariable{⟨*type*⟩}{⟨*name*⟩}{⟨*value*⟩}
\globalsetmfvariable{⟨*type*⟩}{⟨*name*⟩}{⟨*value*⟩}
\globalsetmpvariable{⟨*type*⟩}{⟨*name*⟩}{⟨*value*⟩}
\setmfnumeric{⟨*name*⟩}{⟨*value*⟩}
\setmfpair {⟨*name*⟩}{⟨*value*⟩}
\setmfboolean{⟨*name*⟩}{⟨*value*⟩}
\setmfcolor {⟨*name*⟩}{⟨*value*⟩}

These formerly internal MFPIC macros can be use to define symbolic names for any META-FONT or METAPOST variable type. The last four are abbreviations for the first used with an appropriate value for ⟨*type*⟩. For example, \setmfvariable{pair}{X}{(2,0)} can be abbreviated \setmfpair{X}{(2,0)}. Note that these overwrite any variable with the specified ⟨*name*⟩. For certain internal names, METAFONT will issue an error, but usually the variable is silently redefined.

The commands \setmpvariable and \globalsetmpvariable (note the mp instead of mf) are just alternative spellings . You can use either spelling with either the metafont or metapost option.

---

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

The ⟨*value*⟩ must be a constant of the appropriate type or a METAFONT expression returning the appropriate type. It can also be (or include) other variables previously defined. The \setmfcolor command has been enhanced so that in recent METAPOST the ⟨*value*⟩ can be any of the three types of colors METAPOST allows: numeric (for grayscale color), rgbcolor or cmykcolor. The data type of ⟨*value*⟩ will be examined, and the variable ⟨*name*⟩ will be declared to be a variable of the appropriate type. The same is true of \setmfvariable{color}.

As an example of their use, since dimensions are numeric data types in METAFONT, the command

```
\setmfnumeric{my_spc}{5pt}
\setmfnumeric{my_dia}{.8pt}
```

would set the METAFONT variables my_spc and my_dia to the values 5pt and .8pt, respectively. After that, these variables can be used in any *drawing* command where a dimension is required:

```
\plot[my_dia,my_spc]{Triangle}\rect{(0,0),(1,1)}
```

will plot the rectangle with small triangles of diameter .8pt, spaced 5pt apart.

The knowledgeable user may realize that path and picture are METAFONT data types, and may want use them in \setmfvariable. It is also true that at some level, MFPIC figure macros produce a path and \mfpimage produces a picture. However, MFPIC commands cannot be used in the value portion of \setmfvariable. The TeX code that most MFPIC commands produce would be meaningless to METAFONT. You can store the path produced by figure macros with \store, and store pictures in variables with \mfpimage or even \tile.

With the obvious exception of the \globalsetmfvariable command, these commands define the variable locally. That is, the variable will revert to any previous definition (or become undefined) at the end of the mfpic environment it is defined in. It is in fact local to any METAFONT group. In MFPIC, only \connect ... \endconnect, \mfpimage ... \endmfpimage, and \mfpic ... \endmfpic create METAFONT groups in the graph file.

A warning about variable names. METAFONT and METAPOST allow multi-part variable names like 'arrowhead length' or 'X.r' The part after the first space or '.' is called a suffix. In META-FONT, variable settings are global unless explicitly made local. The code of the \set... commands does make the variable setting local. However, METAFONT syntax forbids this localization when a variable name has a suffix. Moreover, if you localize a variable, METAFONT will localize all variables with that name plus any suffix. Even more, localizing a variable renders all variables with the same name plus suffix locally undefined. The command \globalsetmfvariable simply omits the localization part, so suffixes are permitted, but it cannot 'globalize' something that has previously been localized within the same group.

For example, suppose you use the example code in subsection 4.4.3 and define a custom arrowhead path myAH and the corresponding clearing path myAH.clear. Suppose now you try to make this head the default for the \arrow command by doing the following.

```
\setmfvariable{path}{Arrowhead}{myAH}
```

Then this assignments is local and makes Arrowhead.clear undefined (locally). You cannot use \setmfvariable to define Arrowhead.clear; that will produce an error from METAFONT. You need to do

```
\setmfvariable{path}{Arrowhead}{myAH}
\globalsetmfvariable{path}{Arrowhead.clear}{myAH.clear}
```

and *both* assignments will be local. To make both assignments global, use the global version in both.

\patharr{⟨*name*⟩}...\endpatharr

This pair of macros, acting as an environment, accumulate all enclosing paths, in order, into a path array named ⟨*name*⟩. 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. For this reason, the definitions it makes are 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}.

\setarray{⟨*type*⟩}{⟨*var*⟩}{⟨*list*⟩}
\globalsetarray{⟨*type*⟩}{⟨*var*⟩}{⟨*list*⟩}
\pairarray{⟨*var*⟩}{⟨*list-of-points*⟩}
\numericarray{⟨*var*⟩}{⟨*list-of-numbers*⟩}
\colorarray{⟨*var*⟩}{⟨*list-of-colors*⟩}
\rgbcolorarray{⟨*var*⟩}{⟨*list-of-rgbcolors*⟩}
\cmykcolorarray{⟨*var*⟩}{⟨*list-of-cmykcolors*⟩}

These enable the simultaneous definition of 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. That is, \polyline{X[1],X[2]} and \polyline{X 1,X 2} are the same as \polyline{X1,X2} to METAFONT. 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. For all these array commands, the variable X by itself (not followed by any digit or brackets) becomes a numeric variable equal to the number of elements in the array. Except for \globalsetarray, the arrays are defined locally if these commands occur in an mfpic environment, global otherwise.

Array variables may be used only where the values are processed only by METAFONT or META-POST, they are unknown to TEX. In particular, they cannot be used in commands that position text unless mplabels is in effect. Variables may be used in the ⟨*list*⟩ parameters of commands, but they must have been previously defined or otherwise known to METAFONT.

Since arrays must all be variables of the same type, one cannot mix rgb and cmyk colors. The \colorarray command requires rgb colors (for compatibility with early METAPOST).

Several commands in MFPIC define arrays of objects that can be used in other commands. The main ones are \regpolygon, \piechart and \barchart. These arrays are always global (either because their use is restricted to an mfpic environment or for backward compatibility with the time when they were so restricted).

Using \regpolygon{⟨*num*⟩}{X}{...}{...} causes a pair array named X to be defined having ⟨*num*⟩ elements (and the additional pair X0 for the center). This is in addition to creating the actual figure. The variable X alone becomes a numeric equated to ⟨*num*⟩.

Using \piechart (or \mfppiechart) causes the following arrays to become defined (or redefined):

- piewedge, a path array describing the wedges of the chart. To access piewedge[1], for example, one could use \mfobj{piewedge[1]}. This is almost exactly the same as the MFPIC command \piewdge{1} without optional arguments.
- pieangle, a numeric array, gives the starting angles of the wedges.
- piedirection, a pair array, gives the unit vectors pointing from the center of the piechart through middles of the wedges. For example, if \pieangle1 is 0 and \pieangle2 is 90 degrees, then piedirection1 is $(\cos 45, \sin 45)$, the unit vector whose angle is 45 degrees.

Using \barchart (or \mfpbarchart or any of its aliases) causes the following arrays to become defined (or redefined). The exact meaning depends on whether bars are horizontal or vertical. The following describes horizontal bars; exchange the roles of *x* and *y* if they are vertical (also change 'right' to 'top', etc.):

- barstart, a numeric array, gives the position on the *y*-axis of the leading edge of the bars.
- barbegin, numeric, gives the *x*-coordinate of the leftmost end of the bars.
- barend, numeric, gives the *x*-coordinate of the rightmost end of the bars.
- chartbar, a path array, gives the actual bars. For example, chartbar2 is the rectangle with opposite corners (barbegin2,barstart2) and (barend2,barstart2+barwd), where the numeric variable barwd is the thickness of the bar (which is a height for horizontal bars).
- barlength, the same as barend. This is for backward compatibility; the name was chosen at a time when all the bars had one side on an axis.

4.12.5   Miscelaneous pair expressions

A useful METAFONT operator that produces points is the intermediation operator, whose syntax is

(⟨*num*⟩)[⟨p_1⟩,⟨p_2⟩]

That is, a number or numeric expression in parentheses followed by literal brackets (this is *not* an optional argument) containing two points or pair expressions separated by a comma. It returns an intermediate point on the line through ⟨p_1⟩ and ⟨p_2⟩. The formula for the returned value is $p_1 + \langle num \rangle (p_2 - p_1)$. The midpoint is obtained with ⟨*num*⟩ = .5. If the ⟨*num*⟩ is a pure number, the parentheses can be omitted, but they are required if it is any other numeric expression. Values of ⟨*num*⟩ larger than 1 or less than zero produce points on the line that lie outside the segment from p_1 to p_2. This operator can also be applied to numbers or (in METAPOST) to colors (of the same type). So that (2/3)[3,6] = 5 and .7[green,blue] = (0,.3,.7). See section 4.3 for a description of colors in METAPOST and METAFONT.

pathpoint(⟨*frac*⟩,⟨*name*⟩)

This is another useful METAFONT command. It requires a number, ⟨*frac*⟩, and the *name* of a previously defined METAFONT path variable. (Defined, for example, using \store; see subsection 4.10.2). It returns the point on the path that is approximately that fraction of the path's length from the start of the path. For example to draw a line from $(0,0)$ to the midpoint of an arc, do the following:

```
\store{myarc}\draw\arc{(1,0),(0,2),90}
\polyline{(0,0), pathpoint(.5,myarc)}
```

METAFONT has no general command for calculating the lengths of paths; METAPOST does, but it is quite slow. Thus neither program has an efficient method for finding the described point, so MFPIC uses METAFONT/METAPOST macros that are faster, but less accurate than they could be. Still, the results should (except in pathological cases) be accurate to within a couple of percent of the length of the path. If they are not, adjust the value of the fraction. These remarks about accuracy also hold for any other command (such as \partpath in subsection 4.4.2) that take the fraction of a path length as a parameter.

The pathpoint command is not a basic METAFONT command, but is defined by the GRAFBASE macros that accompany MFPIC.

METAFONT pairs can conveniently be viewed as complex numbers. So grafbase also contains some functions useful in complex analysis (my research field). In what follows a, z and w denote pair variables or constants, and each function interprets them as complex numbers. Also t denotes an angle in radians. There are both numeric and pair valued functions, the type of each is noted after the description:

Arg z   The principle argument of z in radians (numeric).
Log z   The principle logarithm of z (pair).
cis t $(\cos t, \sin t)$, same as dir degrees(t) (pair).
zexp w   The complex exponential, e^w (pair).
zsqrt w   The (principal) complex square root: that z with $-\pi/2 \leq \mathrm{Arg}\, z \leq \pi/2$ and $z^2 = w$ (pair).
sgn z   The signum, $\mathrm{sgn}(0,0) = (0,0)$ otherwise $\mathrm{sgn}\, z = z/|z|$ (pair).
conj z   The complex conjugate, $\bar{z}$ (pair).
Moebius(a) z   The Möbius transformation $(z+a)/(1+\bar{a}z)$ (pair)
pshdist(z,w)   The pseudohyperbolic distance between z and w: $|z-w|/|1-\bar{w}z|$ (numeric).
kelvin(z)   The Kelvin transform $1/\bar{z}$ (pair)

METAFONT will happily add and subtract pairs but to multiply and divide complex numbers requires new operations. These are given by (z zmul w) and (z zdiv w). They operate on pairs and produce pairs.

4.12.6 MANIPULATING METAFONT PICTURE VARIABLES

\tile{⟨*tilename*⟩,⟨*unit*⟩,⟨*wd*⟩,⟨*ht*⟩,⟨*clip*⟩}
  ⟨MFPIC *drawing commands*⟩
\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. Actually, a tile is a composite object. After \tile{Nick, ... } ... \endtile a picture variable Nick.pic is created as well as numeric variable Nick.wd and Nick.ht. These are needed by the \tess command, below.

The units of drawing are given by ⟨*unit*⟩, which should be an explicit 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 the 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 see the warning about memory use by the \tess command. The ⟨*tilename*⟩ is globally defined by this command.

\tess{⟨*tilename*⟩}...

This rendering macro tiles the interior of a closed path with a tessellation comprised of copies of the *tile* specified by ⟨*tilename*⟩. The tile must have been previously created by \tile{⟨*tilename*⟩,

`...` }. Tiling an open curve is technically an error, but the METAFONT code responds by drawing the path and not doing any tiling. The METAFONT code places shifted copies of the tile picture in a rectangular grid sufficient to cover the region, then clips it to the closed path before drawing it.

Tiling large regions with complicated tiles can exceed the capacity of some versions of META-POST. 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.

Before version 0.8, `\tile` was the only way to create a picture variable, and the only way to draw this picture was with the `\tess` command. Now we have the following command to place multiple copies of a picture:

`\putmfpimage{⟨name⟩}{⟨list⟩}`

This take the name of a picture variable and copies the picture at each location in the ⟨*list*⟩, which should be a comma-separated list of coordinate pairs in graph coordinates. The picture is copied so that its *reference point* is placed at each of the locations. The reference point of a picture created with `\tile` is its lower left corner.

`\mfpimage[⟨refpt⟩]{⟨picname⟩}`
  ⟨MFPIC *drawing commands*⟩
`\endmfpimage`

This is another way to create a picture variable. The drawing commands within the `mfpimage` environment contribute not to the current MFPIC picture, but rather to the picture variable named in ⟨*picname*⟩. Otherwise, they operate exactly as they would outside this environment, using the same coordinate system and the same default values of all parameters, etc. (unlike the `tile` environment, which defines its own coordinate system). The picture is created with its reference point at the point ⟨*refpt*⟩ given in the optional argument. The default is (0,0). For example:

```
\mfpimage[(1,1)]{Jan}
  \fill\rect{(0,0),(1,1)}
  \fill\rect{(1,1),(2,2)}
  \rect{(0,0),(2,2)}
\endmfpimage
```

produces a simple 2-by-2 chessboard with its reference point at the center point $(1, 1)$. One can then write something like

```
\putmfpimage{Jan}{(1,1),(3,1),(1,3),(3,3)}
```

to get a 4-by-4 chessboard: the picture Jan copied with its center at each of the listed points.

The behavior of `\tlabel` in an `mfpimage` environment depends on the setting. If mplabels is turned off, then labels are added by TEX and are *not* included as part of the named METAFONT or METAPOST picture variable. They are placed on the current picture as if the `mfpimage` environment were not there at all. If mplabels is turned on and overlaylabels is also turned on, or if the `mfpimage` environment is between `\startbacktext` and `\stopbacktext`, then the labels will be saved and placed when the `mfpic` environment ends and *not* added to the named picture variable. Thus, to include text labels in the named picture variable, you must have mplabels on, overlaylabels off, and `mfpimage` outside any `\startbacktext`/`\stopbacktext`.

The picture created by `\mfpimage` is locally defined. That is, it becomes undefined at the end of the current `mfpic` environment. If one needs it to be global, one can use `\globalsetmfvariable` (see subsection 4.12.4) to copy it to another variable. For example. the command

```
    \globalsetmfvariable{picture}{Dan}{Jan}
```
would make Dan globally defined to be equal to the current value of the picture Jan. Note that picture variables can consume a lot of METAFONT's memory. Copying one variable to another doubles the amount of memory, at least until the end of the mfpic environment.

You can use \putmfpimage inside a mfpimage environment, provided the picture being placed has been previously defined. Nesting a mfpimage inside another has not been tested at all and so is not recommended. But if it works, the inner image would be local to the environment created by the outer one, and so would be of limited use. One can use the LATEX environment construct \begin{mfpimage} ... \end{mfpimage} in a LaTeX document instead of \mfpimage ... \endmfpimage.

4.12.7 METAFONT LOOPS

All the MFPIC loop commands create a loop (in the METAFONT language) in the output file. The METAFONT commands in that loop are executed repeatedly by METAFONT or METAPOST. From the point of view of TEX, however each command occurs only once. Starting with version 0.9, these loops can be created inside or outside the mfpic drawing environment. If outside, they must not contain any drawing commands, but can contain commands that set variables, perform computations, etc.

\mfpfor{⟨*for-loop header*⟩}
  ⟨MFPIC *commands*⟩
\endmfpfor

This creates a for-loop in the METAFONT output file. The \mfpfor writes the start of the loop and \endmfpfor writes the end. Any code written in the output file between them is executed repeatedly by METAFONT, according to the information in ⟨*for-loop header*⟩. There are two types of headers possible, illustrated by the following examples.

```
    \mfpfor{center = (0,0), (1,0), (0,1)}
      \gfill\circle{center,1}
    \endmfpfor
```

This example will fill three circles of radius 1 with centers at the three given points. This type of header has the format

  ⟨*variable*⟩ = ⟨*list*⟩

where ⟨*variable*⟩ should be a simple variable name and ⟨*list*⟩ is a comma separated list of items of the appropriate data type. In the above, center is equated to pairs, but in the following

```
    \mfpfor{radius = 1,3,4}
      \dotted\circle{(0,0),radius}
    \endmfpfor
```

radius gets numeric values.

The other type of header uses a stepped variable:

```
    \mfpfor{level = 3 step 2 until 9}
      \circle{(0,0),sqrt(level)}
    \endmfpfor
```

This will cause the METAFONT variable level to step through the values 3, 5, 7 and 9 and the circles with radius $\sqrt{3}$, $\sqrt{5}$, etc. will be drawn. This type of header has the format

  ⟨*variable*⟩ = ⟨*start*⟩ step ⟨*delta*⟩ until ⟨*stop*⟩

where ⟨*variable*⟩ is as before, while ⟨*start*⟩, ⟨*delta*⟩ and ⟨*stop*⟩ are numeric values. If ⟨*delta*⟩ is positive the loop is skipped entirely if ⟨*stop*⟩ is less than ⟨*start*⟩. Otherwise the loop is executed successively with the variable equal to ⟨*start*⟩, then ⟨*start*⟩ + ⟨*delta*⟩ then ⟨*start*⟩ + 2⟨*delta*⟩, etc., as long as the variable is not greater than ⟨*stop*⟩. The behavior is similar if ⟨*delta*⟩ is negative, except the loop is repeated only as long as the variable is not less than ⟨*stop*⟩. If ⟨*delta*⟩ is 0, then the METAFONT run will generate an error.

Note that the index variable (`center` and `radius` in the above two examples) is a temporary METAFONT variable. If mplabels is turned on, this variable will work as expected in the *location* parameter of a `\tlabel` command, but if it is used in the *label* part, it will be interpreted as TEX code and printed as is. The index variable reverts to its previous state outside the loop. That is, if it existed before the loop, it regains its previous value after the loop, and if it was undefined before the loop, it is again undefined after.

The single word 'upto' can be used as an abbreviation for "`step 1 until`" and 'downto' for "`step -1 until`" in for-loop headers. Spaces are not significant in for-loop headers, except to distinguish the keywords (e.g. `step`) from variable names that might be used (e.g., for ⟨*start*⟩).

`\mfpwhile{`⟨*condition*⟩`}`
  ⟨MFPIC *commands*⟩
`\endmfpwhile`

The ⟨*condition*⟩ should be an expression that can be either true or false about a METAFONT variable that changes at some time during the loop body. The loop body is executed (by METAFONT) as long as the condition is true. Example:

```
\setmfvariable{numeric}{R}{20}
\mfpwhile{R > 1}
  \rect{(0,0), (R,3R)}
  \mfcmd{R:=R/2}
\endmfpwhile
```

There are no MFPIC commands to *systematically* change a variable, so in this example we have resorted to directly writing a METAFONT assignment command via `\mfcmd` (see subsection 4.12.3 above) that reduces R by half. The loop will be executed with R having the successive values 20, 10, 5, 2.5, and 1.25. The resulting picture could have been achieved with `\mfpfor` using this list of values.

`\mfploop`
  ⟨MFPIC *commands*⟩
`\mfpuntil{`⟨*condition*⟩`}`
  ⟨MFPIC *commands*⟩
`\endmfploop`

The body of this loop will be repeated until the ⟨*condition*⟩ becomes true. The condition should be some expression that can be either true or false about a variable that changes during the loop execution. It should eventually become true. If an mfploop environment does not contain an `\mfpuntil` command, then the `\endmfploop` command will generate a warning message. If the warning is ignored, and the user has not otherwise arranged for loop termination,[19] the `.mf` file will contain an infinite loop. The `\mfpuntil` command will break the loop at whatever point it occurs. Example:

---

[19]Perhaps by means of `\mfsrc` commands. It is because of this possibility that only a warning is produced and not an error. If the warning becomes annoying, adding `\mfpuntil{false}` to the loop will silence it. This command will never break the loop because the condition `false` (of course) never becomes true.

```
\setmfvariable{numeric}{R}{20}
\mfploop
  \mfcmd{R:=R/2}
  \mfpuntil{R <= 1}
  \rect{(0,0), (R,3R)}
\endmfploop
```

This will draw rectangles with *R* equal to 10, 5, 2.5, and 1.25. On the next execution of the loop the condition R<=1 is true, and the break occurs before the next rectangle is drawn. Note that any \mfpwhile could be encoded with \mfploop. In fact, the code written to the output file by

$$\mfpwhile\{\langle condition\rangle\}$$

is identical to that written by

$$\mfploop\mfpuntil\{not\ \langle condition\rangle\}$$

The command \mfpuntil can also be used in mfpfor and mfpwhile environments to break the loop prematurely when the given condition becomes true.

All three of these loop structures bracket the inner code in a TEX group. In a LATEX document, the usual \begin/\end style can be used. For example,

```
\begin{mfpfor}{radius = 1,3,4}
  \circle{(0,0),radius}
\end{mfpfor}
```

Just to be clear: in all the examples, what is written to the figure file is a *single* circle or rectangle drawing command, bracketed by code that causes METAFONT to execute it several times with different values for the variable. From TEX's point of view, there is only one MFPIC drawing command.

4.12.8 Miscellaneous

\mfmode{⟨*mode-name*⟩}
\mfresolution{⟨*DPI*⟩}

When working with METAFONT, the code in grafbase.mf needs to know the resolution at which to make the font with all the figures. If the wrong resolution is assumed, the figure may end up appearing wrongly scaled or have other problems (especially with shading). If your DVI viewing/printing program and the file modes.mf are correctly configured, nothing may need to be done. If not, as a last resort, you can set the METAFONT mode or the METAFONT resolution in your .tex file with these commands. If you don't know what that means, ask a guru, but then you should probably be using METAPOST and not METAFONT.

Note that this is a *last resort*. The code in grafbase.mf first checks if mode has been defined, then checks if localfont is defined and only then checks if the resolution has been set by this method (if all three fail, it uses a value of 600 DPI).

\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 that mfpic environment. This is useful if all one wishes to do in the current mfpic environment is to make tiles (see above) or define picture variables with \mfpimage or path arrays with \patharr. While \mfpimage defines the picture locally, one can globally copy it to another variable with \globalsetmfvariable (see subsection 4.12.4).

`\assignmfvalue{`⟨*TEX-macro*⟩`}{`⟨*MF-expr*⟩`}`
`\assignmpvalue{`⟨*TEX-macro*⟩`}{`⟨*MF-expr*⟩`}`
`\globalassignmfvalue{`⟨*TEX-macro*⟩`}{`⟨*MF-expr*⟩`}`
`\globalassignmpvalue{`⟨*TEX-macro*⟩`}{`⟨*MF-expr*⟩`}`

The command names spelled with 'mp' are no different than the ones spelled with 'mf'. You can use either spelling with either the metafont or metapost option.

These commands causes the ⟨*MF-expr*⟩ to be written to the output file for METAFONT to evaluate. The resulting value is then written to the `.log` file of that METAFONT run. On the next TEX run, if mfpreadlog (see section 2.11) is in effect, the macro ⟨*TEX-macro*⟩ will be defined to produce the resulting value. For example:

```
\setmfnumeric{s}{2}
\assignmfvalue{\val}{exp s}
\tlabel(1,2){$e^s = \val$}
```

After METAFONT is run and then TEX run a second time, `\val` will acquire the definition '7.38905', the value of `exp s` when s=2 (i.e., e^2, correct to at least the fourth decimal place). If mplabels is in effect, the correct label is written to the figure file only during this second run, and a second METAPOST run will be required. In many cases (when using pdfTEX, for example, or when the label changes the figure dimensions), a third TEX run will be required to make the figure correct when it is included in the document.

Before METAFONT is run to evaluate the expression, the macro produces '???'. Thus, it cannot be used in places where a number is needed (as in the position arguments of a `\tlabel` command). Note also that if a command defined by `\assignmfvalue` is used in a tlabel with mplabels in effect, then mplabels must be in effect during the `\assignmfvalue` command as well.

The 'global' version makes the definition of the ⟨*TEX-macro*⟩ global, surviving the current group. In particular, it can be used in other pictures. The plain versions create commands that are only locally defined. Past versions of this manual stated that you can say

```
\global\assignmfvalue
```

to define the macro globally. This turns out not to be true in all cases. If a global definition is needed, use the global versions above.

Because of the asynchronous nature of the definition process, using `\assignmfvalue` with the same macro name more than once in the same `mfpic` environment will not work. The macro becomes defined upon reading the logfile during the execution of `\opengraphsfile`, and it will end up with the last definition encountered. (The same is true for uses outside `mfpic` environments: the macro acquires the last such definition.) Moreover, the definition is associated to a picture by number. Which means that reordering the environments or changing the numbering by any means will require the TEX-METAFONT-TEX sequence (or more) to be repeated.

If the ⟨*TEX-macro*⟩ is already defined, no warning will be issued and the command will be redefined, so be careful in the name chosen. If mplabels is turned off when `\assignmfvalue` is used, but turned on before the ⟨*TEX-macro*⟩ is used in a `\tlabel` command, the macro definition will not be written to the `.mp` file, and either an error message, or incorrect label will result when METAPOST tries to make the tlabel.

The concept and much of the code for `\assignmfvalue` came from Werner Lemberg. However, I have rewritten it substantially to conform to MFPIC conventions and so any errors are my responsibility.

`\cutoffafter{⟨obj⟩}`...
`\cutoffbefore{⟨obj⟩}`...

These prefix macros modify the following path by cutting part of it off. They take an 'object' (a variable in which a path was previously stored using `\store`) and uses it to trim off one end of 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[20]) is cut off. This is reliable only when there is a unique point of intersection.

These macros can be used to create a curve that starts or ends right at another figure without having to know the point where the two curves intersect.

`\randomlines{⟨maxshift⟩}`...
`\randomizepath{⟨maxshift⟩, ⟨weirdness⟩}`...

These modify the following path by applying random shifts to the nodes of a path. The first one, `\randomlines` then simply connects those new points by straight lines, while the second one also applies randomization to the control vectors. The ⟨*maxshift*⟩ argument is either a positive number (in graph units) that limits the distance a node can be moved, or it is an ordered pair of positive numbers, in which case the first limits the horizontal distance and the second limits the vertical. If ⟨*maxshift*⟩ is larger than the distance between nodes, cusps or loops are likely in the result.

For `\randomizepath` the ⟨*weirdness*⟩ parameter controls how the control vectors are modified. Roughly speaking the control vectors are randomly rotated up to $30⟨weirdness⟩$ degrees and randomly scaled up or down by a factor of $2^{⟨weirdness⟩}$. (A "control vector" is a vector pointing from a node to one of its control points.) However, this is done in a way that preserves smoothness at each node where the path is smooth. Values of ⟨*weirdness*⟩ greater than 1 are probably much too weird.

`\brownianmotion{⟨start⟩,⟨num⟩,⟨scale⟩}`
`\browniangraph{⟨num⟩,⟨scale⟩}`
`\randomwalk{⟨start⟩,⟨num⟩,⟨scale⟩}`

These figure macros build a few standard kinds of random paths. The `\brownianmotion` path starts at the point ⟨*start*⟩, then proceeds in a straight line in a random direction a random distance. This is repeated ⟨*num*⟩ times to form a polyline. The random process used is a Gaussian in each coordinate, scaled so that the random distance has a standard deviation equal to ⟨*scale*⟩. Thus, ⟨*start*⟩ is a coordinate pair in graph coordinates, ⟨*num*⟩ is a positive whole number and ⟨*scale*⟩ is a positive real number (in graph units). In rare cases, the random distance can become quite large, but on average it will be about $0.56 \times ⟨scale⟩$. The size of the resultant path (its bounding box) can also be, in rare cases, quite large, but it is usually on the order of $\sqrt{⟨num⟩}$ times ⟨*scale*⟩.

The second path, `\browniangraph`, represents the graph of a one-dimensional Brownian motion. It is random only in the vertical direction as the rightward motion represents the uniform passage of time. It starts at $(0,0)$ and the ⟨*scale*⟩ is both the constant rightward step as well as the standard deviation for the *y*-coordinate. Users will need to apply a shift if they want to change the starting point, and a vertical scaling if they want a scale factor different from the step size.

Despite their names, the paths produced are technically not Brownian motion, but rather 'Gaussian random walks'. However, for small ⟨*scale*⟩ and large ⟨*num*⟩ they can be used to approximate Brownian motion.

---

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

Finally, \randomwalk is just like \brownianmotion except that only the direction is random. The distance is always equal to ⟨*scale*⟩.[21]

There can be a problem with the size of ⟨*num*⟩ in these three macros. Numbers greater than a certain METAFONT/METAPOST parameter called max_points (see the discussing at \levelcurve in section 4.6.2) will produce an error from METAFONT or METAPOST. But also sharp turns will take up space in something called the *rounding table*. This has no bearing on METAPOST, and in META-FONT it only matters if the parameter autorounding is positive. MFPIC leaves autorounding at the default of 2, since this value makes drawings in METAFONT look best. In this case, the value of ⟨*num*⟩ should be less than about 500.

\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 type-setting 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. MFPIC currently uses font dimensions 23 through 52, which should be OK.

Almost all of MFPIC's basic dimension parameters are font 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 5.6.3 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 '.'

---

[21]This is only one kind of an infinite variety of possible random walks. See http://en.wikipedia.org/wiki/Random_walk for a discussion. MFPIC implementation of other kinds is left to the interested user (for example, using \turtle with random displacements).

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 5.6.3 for further dicussion.

\setfilenametemplate{⟨*template*⟩}

With the metapost option, when you write \opengraphsfile{figs}, a file figs.mp is created. By default, running METAPOST on it results in files named figs.1, figs.2, etc. Recent META-POST allows the output filenames to be modified. As of MFPIC version 1.00, you can do this to some extent from your .tex file. One needs to define a template that tells METAPOST how to construct the output file name from the 'jobname' and the figure number. This is done with the above command. In ⟨*template*⟩ you can put any plain characters, plus the two special tokens: _ and \#. Each figure's filename is constructed by replacing these tokens with the METAPOST jobname and the figure number, respectively. For example, with the jobname figs,

    \setfilenametemplate{my_-\#.mps}

will cause the figure files to have names myfigs-1.mps, myfigs-2.mps, etc., instead of the defaults. MFPIC adjusts the definition of \setfilename accordingly, so that the correct filenames are used.

Do not use this command unless you know your version of METAPOST is recent enough to have this capability. Under the metafont option, this command is simply ignored, but MFPIC has no way of checking the METAPOST version on its own.

If you are using LATEX, the \includegraphics command requires that the included figure file be recognized as METAPOST output. In practice, this usually means its extension *must* be .mps. As an exception, it may also be the current figure number (the default if \setfilenametemplate is not used), because MFPIC has always arranged for that to be recognized. The user may also issue a \DeclareGraphicsRule command to get other extensions recognized. See the documentation of the GRAPHICS package.

\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 '110' because the version is 1.10. It can be used to test the version:

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

\mfpicversion was added in version 0.7.

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, any \identity commands are expanded and only the contents appear in the output. (TEX parses the argument to assign meanings to \Ax and \Ay.)

If the BABEL package is loaded with certain options, the comma may become a special character. In that case, one may need to deactivate babel shorthands before some MFPIC code. One might use \everymfpic to do this in every mfpic environment. In some cases, one may need to reactivate babel shorthands insided \tlabel, and one might use \everytlabel for this purpose. See your BABEL documentation for the commands to do these things.

5 Appendices

5.1 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, Werner Lemberg, for the \assignmfvalue command, and Zaimi Sami Alex for suggestions.

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

5.2 Changes History.

See the file changes.txt for a somewhat sporadic history of changes to MFPIC. See the file README for changes added since the previous version, and for any known problems.

5.3 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 raggedcaptions	\usecenteredcaptions, \nocenteredcaptions \useraggedcaptions, \noraggedcaptions	If command is used inside an `mfpic` environment, it should come before the \tcaption command.
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
mfpreadlog	\mfpreadlog	Needed for \assignmfvalue. Must occur before \opengraphsfile.

5.4   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 patterns named `dashtype0` through `dashtype5`, or predefined colors named `colortype0` through `colortype7`, or predefined symbols named `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.) Four of the eight colors use the cmyk model when the METAPOST version is at least 1.000.

Name	Color	(R,G,B)	(C,M,Y,K)
colortype0	black	(0,0,0)	(0,0,0,1)
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)	(0,.85,0,.15)
colortype6	cyan	(0,.85,.85)	(.85,0,0,.15)
colortype7	yellow	(.85,.85,0)	(0,0,.85,.15)

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

5.5 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 4095 pixels is. In fonts prepared for a LaserJet4 (600 DPI), this means 6.825 inches (17.3355cm). For a 1200 DPI pronter, the limit is 3.4125 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 $4.5 * 4.5 = 20.25$: after multiplying by the \mfpic scaling factor, the \mfpicunit in inches, and the DPI value, this produces $20.25 \times 10 \times 0.03937 \times 600 > 4783$ pixel units. 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. Thus, the fact that this particular picture was clipped to a much smaller size for printing did not help.

In METAPOST, the limit on numeric values is only 8 times as high: 32768. However, that is independent of printer resolution and is interpreted as POSTSCRIPT points (TEX's 'big points'). At 72 points to the inch, this allows figures to be about 12.64 yards (11.56 m).

5.6 Special Considerations When Using METAPOST.

5.6.1 REQUIRED SUPPORT

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

TEX format	support needed
plain TEX	The file epsf.tex or epsf.sty
LATEX209	(No longer supported, but plain TEX methods might work)
LATEX	The package GRAPHICS or GRAPHICX
pdfLATEX	The package GRAPHICS or GRAPHICX with option pdftex
plain pdfTEX	The files supp-pdf.mkii or supp-pdf.tex and (possibly) supp-mis.tex
In all cases	The files grafbase.mp, dvipsnam.mp and mfpicdef.tex 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. If METAPOST cannot find the file `grafbase.mp`, then by default it will try to input `grafbase.mf`, which is generally fatal (and always futile).

The remaining files should be in directories searched by the appropriate TEX variant. The file `mfpicdef.tex` is input by TEX when METAPOST is processing labels in `.mp` files created by MF-PIC. The user is free to add commands of his own to that file, but be warned that updates to MF-PIC will overwrite it. Better to create ones own file (say `mydefs.tex`) and arrange its input via `\mfpverbtex{\input mydefs.tex}`

In case pdfLATEX is used, the GRAPHICS package is given the pdftex option. This option requires the file `pdftex.def` which currently inputs one of the `supp-pdf` files. Early versions of `supp-pdf.tex` will input `supp-mis.tex`. These three files should be supplied with most TEX installations.[22] 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. MFPIC will load whichever one it decides is required. In the LATEX 2_ε case, MFPIC will load the GRAPHICS package. If the user wishes GRAPHICX, then that package must be loaded before MFPIC.

5.6.2   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 chokes 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`, some 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, sub-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 (except in horizontal hatching), 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 along the boundary curve(s), 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 dashed outline.

---

[22]They are part of the ConTEXt distribuition. At this writing, these files, plus a few others, can also be found at `CTAN/graphics/metapost/contrib/tools/mptopdf/tex/context/base/`.

5.6.3   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:

plain TEX	\def\setmfpicgraphic#1{\epsfbox{#1}}
LATEX209	(No longer supported, but likely the plain TEX definition will be selected.)
LATEX	\def\setmfpicgraphic#1{\includegraphics{#1}}
pdfLATEX	\def\setmfpicgraphic#1{\includegraphics{#1}}
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 2$_\varepsilon$.
>   \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. A good deal of MFPIC's figure placemant code assumes that the size of the figure is consistent with the coordinate system set up by the \mfpic command. With mplabels plus truebbox it might 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.[23]

---

[23]Unless modified by \setfilenametemplate, of course. See subsection 4.12.8.

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.[24] 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.

5.7 MFPIC **and the Rest of the World.**

5.7.1 THE LITERATURE

This author has personal knowledge of one mathematical article which definitely uses MFPIC to create diagrams, and that is this author's joint paper with J. Duncan and C. M. McGregor: *On the value of pi for norms in* $\mathbf{R}^2$ in the College Mathematics Journal, vol. 35, pages 84–92. Oddly enough, it was McGregor and not I who chose to use MFPIC for the illustrations.

There also exists a book that makes use of MFPIC: *Introduction to functional equations: theory and problem solving strategies for mathematical competitions and beyond* by Costas Efthimiou, MSRI/Mathematical Circles Library, vol. 6, 2011.

There are at least two major publications 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 LATEX Companion* (Second Edition) mentions MFPIC, but only in its annotation of the bibliography entry for TEX *Unbound* (see below).

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[25] 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{(`$\langle x\rangle$`,`$\langle y\rangle$`),` $\langle radius\rangle$`,` $\langle angle1\rangle$`,` $\langle angle2\rangle$`}`

The macro `\plr{(`$\langle r_0\rangle$`,`$\langle\theta_0\rangle$`),(`$\langle r_1\rangle$`,`$\langle\theta_1\rangle$`),...}` is now used to convert polar coordinate pairs to rectangular coordinates and 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 `\plrcyclic`, `\plrlines` and `\plrpoint`.

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

---

[24]But appropriate use of `\numexpr` (in eTEX) for calculations is probably OK.

[25]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.

\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, the LaTeX template on page 496 is no longer the only possiblity: recent MFPIC may be loaded with \usepackage.

5.7.2 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 together 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 (perhaps) DPIC to produce the output. The MFPIC code produced appears to be compatible with the current MFPIC.

5.8 Index of commands, options and parameters.

5.9 List of commands by type.

5.9.1 FIGURES

\arc, 21
\axis, 16
\axisline, 16
\belowfcn, 44
\border, 16
\brownianmotion, 75
\browniangraph, 75
\btwnfcn, 43
\btwnplrfcn, 43
\cbeziers, \closedcbeziers, 64
\chartbar, 25
\circle, 20
\computedspline,
  \closedcomputedspline, 63
\convexcurve, \closedconvexcurve, 24
\convexcyclic, 24
\cspline, \closedcspline, 63
\curve, \closedcurve, 23
\cyclic, 23
\datafile, 46
\DEgraph, 45
\DEtrajectory, 45
\ellipse, 22
\fcncurve, 24
\fcnspline, 63
\fullellipse, 22
\function, 43
\ganttbar, 25
\gbrace, 26
\graphbar, 25
\halfellipse, 22
\histobar, 25
\hypergeodesic, 23
\levelcurve, 44
\lines, 15
\mfbezier, \closedmfbezier, 64
\mfobj, \mpobj, 57
\parafcn, 43
\periodicfcnspline, 63
\piewedge, 26
\plrfcn, 43
\plrregion, 44
\polygon, 15
\polyline, 15
\pshcircle, 23

\qbeziers, \closedqbeziers, 64
\quarterellipse, 22
\qspline, \closedqspline, 63
\randomwalk, 75
\rect, 15
\regpolygon, 15
\sector, 22
\tlabelcircle, 54
\tlabelellipse, 54
\tlabeloval, 54
\tlabelrect, 53
\turtle, 24

5.9.2 RENDERINGS

\areagradient, 40
\corkscrew, 37
\dashed, 35
\dotted, 36
\doubledraw, 35
\draw, 35
\gclear, 38
\gclip, 38
\gendashed, 36
\gfill, 38
\gradient, 39
\hatch, 39
\lhatch, 39
\plot, 36
\plotdata, 48
\plotnodes, 36
\polkadot, 38
\radialgradient, 40
\rhatch, 39
\sinewave, 37
\shade, 38
\tess, 69
\thatch, 39
\xhatch, 39
\zigzag, 37

5.9.3 ARROWS

\arrow, 33
\arrowhead, 34
\arrowmid, 34
\arrowtail, 34

5.9.4   MODIFYING FIGURES

5.9.5   LENGTHS

5.9.6   COORDINATE TRANSFORMATION

5.9.7   SYMBOLS, AXES, GRIDS, MARKS