Creating Pictures in LaTeX with METAFONT and METAPOST

Volkmar Liebscher

March 17, 2000

1 Introduction

Because of the defenciencies of PiCTeX I got the idea for using METAFONT to create pictures for use inside TeX or LaTeX came to me in the process of making pictures (i.e. graphics and labels) for geometric proofs. It was to long and boring for me to calculate all points, angles, functions explicitely by hand or pocket calculator. On the other hand, METAFONT is a useful language to describe (plain) geometric details, e.g. for the midpoint between to points z_1 and z_2 use the description

$$0.5[z_1, z_2]$$

which is both short and handy. Try anything like this in PiCTeX! I wanted to get anything like a logical description of pictures in LaTeX to make the line "write down the logical structure, and LaTeX prepares the document" even longer into graphical structures.

There was even another fact: the slowness of PiCTeX. It is so boring to prepare a document with PiCTeX–graphics, that it is almost impossible, if no 486 PC or faster is required. So the idea was, to let METAFONT(METAPOST) do, what METAFONT(METAPOST) can best and TeX (LaTeX) vice versa. I relied on METAFONT because METAFONT is well connected to TeX and so there are possibly no big difficulties in using both of them.

I wanted to try another feature, which is not so good implemented in METAFONT: graphical representation of 3D–objects. `mfpic` only provides Kavalier perspective, some sort of parallel perspective and a perspective which comes from an arbitrary view direction, but I think it would be enough.

Last not least, I want to mention another project also named `mfpic` by Tom Leathrum (`moth@dartmouth.edu\verb`), but the version I found on network was incomplete. May be, we could merge together and create something better.

2 The files.

The `docstrip` source is found in `mppic.dtx`. You may run `initex` on `mppic.ins`.

There are several files: `mppic.sty`, containing the LaTeX–macros for `mppic`, `mpbase.mf`, containing the METAFONT–macros, `mpbase.mp`, containing the METAPOST–macros and `mpdoc.tex`, this file for processing through LaTeX. For some additional advises you can process the file `examp.tex`, which gives some inside

into the utilities of `mfpic`. `mfpic.sty` is an obsolete file as the Package was renamed in version 2.

The Package uses the `graphics` and the `robscan` Package, see the respective `.dtx` files.

3 Installation and Use

You must have both LaTeX and METAFONT (or METAFUN) to use. But it is enough, to install the plain bases for METAFONT and METAFUN respectively. If you want to use shading and hatching macros, be sure, that your METAFONT is big enough. There may be the problem, that some drivers give you memory overflow errors, if you try to get to large graphics (METAFONT allows only pictures arround 36cm×36cm large (about 14in×14in) at 300dpi). Thus may be, you get problems at high resolutions. One possibility to get arround this is to split the picture (see the macro `\dividechar`).

To process the sample file, first run LaTeX on the file `jobname.tex`. Then run immediatedly METAFONT in `localfont` mode on the File `mpjobname.mf`. (Run METAFUN on `mpjobname.mp`. For this You should activate the option `mp` by the `\usepackage[mp]{mppic}` command) On some installations you have to convert the output `.gf` file. (on my installation I had to use `gftopk.exe`). With option `[mfjob]` you get `jobname.mfj` to run through mfjob (may be, you have to customize the font generation to your favourite printer). By running again LaTeX there should be a file `jobname.dvi`, which gives you nice graphics output.

These three steps of processing – processing with TeX, processing with METAFONT/METAFUN, and reprocessing with TeX – may not always be necessary. If you don't change the number and the position of pictures (and labels inside pictures), you can avoid reprocessing with TeX. If don't change the pictures, you needn't run METAFONT again.

There is one source of error to note: If you run LaTeX and stop before the document is complete, then running METAFONT will end in a situation, where METAFONT wants to input something. Then you have to type `bye;` and see, what will happen. It will be good to delete the file `mpjobname.log` (or some other `.log` file, if you changed the name), to avoid failures by TeX if you reprocess after finding the source of the error. This sort of procedure is recommended in any situation, where METAFONT gives an error.

4 The structure of `mfpic`

If you use the package `mppic` by running LaTeX on the file `jobname.tex`, TeX constitutes by `\write` commands the file `mpjobname.mf` (you may change this name by a `\setmpfontname` command). The prefix `mp` is used for to avoid conflicts of `.log`–files. Then you run METAFONT on the file `mpjobname.mf` which will create `mpjobname.log`, `mpjobname.tfm` and `mpjobname.300gf` or so, depending on your final device output. The first file is used for communication between TeX and METAFONT, the second and the third give you the necessary font informations. After another run of LaTeX there will be good `jobname.dvi` file.

5 List of macros and Environments

5.1 Generalities

Our general delimiter is the | symbol, ranges are denoted by *a*--*b*. If *b* is negative, you have to type three -'s. The single elements of a list are separated by commata.

Optional arguments are included in [and], like in the usual LaTeX syntax.

5.2 File handling

\setmpfontname{⟨*fontname*⟩}

Redefines the name of the font original beeing mpjobname to *mpfontname*. Don't set {⟨*fontname*⟩}=\jobname! If there was even output to a .mf file, all settings will be choosen so, that upto this moment the old font is required and from now on the new one is available. But you have to process two (or more) files through METAFONT.

5.3 Environments

\begin{mfpic}{⟨*unitlengths*⟩} [*xmin*--*xmax*|*ymin*--*ymax*|]
\end{mfpic}

This Environment gives you the opportunity to make all picture building operations like drawing, filling and erasing. *unitlengths* stands either for *unitlength* or possibly *xunitlength*—*yunitlength*, the former is handled like *unitlength*—*unitlength* and the lengths are given like in TeX and LaTeX usual (in pt or in or cm or mm ...) You may change the bounds afterwards with \bounds.

\begin{3D}{⟨*unitlengths*⟩}[*xmin*--*xmax*|*ymin*--*ymax*|*zmin*--*zmax* |]
\end{3D}

The same as \begin{mfpic}, but with three parameters instead of two. Any description is three dimensional in this environment (see description of commands.)

\begin{object}{⟨*objectname*⟩}[⟨*list of parameters*⟩]
\end{object}

This defines an object, i.e. a series of picture building commands, under the logical name *objectname*. You don't need to be inside a picture. *list of parameters* is a list of names such that you can set this parameter by \name{*parametersetting*} to be the same as *parametersetting* (see section about Parameters). These parameters are the only nonnumeric constants you can refer to inside the object. Later on you can insert this object into any picture with different parametersettings. The parameters from *list of parameters* get at the time of use (in the TeX level) their actual value. Inside an object there are allowed all commands from the mfpic environment besides that commands, that use the picture bounds.

\begin{rel}
\end{rel}

environments:3D,mfpic,object

Inside this environment you can put down in METAFONT–like syntax relations between points and paths (the latter only in the environments mfpic and object. You have to mask logical names declared as points via a points definition (see macros below) with an @ (and if the name is not a single letter, you have to insert { and }). Likewise paths with ?. Inside the \begin{3D} environment you have

even to mask explicite points with an @ (unfortunately, but METAFONT is not very good prepared to handle threedimensional things). You can also use the explicite `\mppoint,\mppath,\tdpoint,\tdpath` depending wether you are 2D (mp...) or 3D (td...).

Relations are given by `RHS=LHS`, where `RHS` and `LHS` are both linear expressions in explicite and implicite points by using METAFONT syntax (see section below).

By putting down the relation, you must assure, that the values you refer to are uniquely determined.

Make sure, that you use inside `rel` no other environment.

`\begin{2Dtext}`
`\end{2Dtext}`
environments:`mfpic,object`

Inside this environment the masking conventions described above are active.

`\begin{3Dtext}`
`\end{3Dtext}`
environments:`3D`

The same, but you have also `\xpart,\ypart,\zpart` to define something coordinatewise.

5.4 Declarations

Declarations can be made in all environments exept the `rel` environment.

`\points{`⟨*list of points*⟩`}`
environments:`mfpic,3D,object`

Declares some logical synonym for a point, i.e. *list of points* is a list of names (consisting of normal characters and digits), divided by commata.

`\paths{`⟨*list of paths*⟩`}`
environments:`mfpic,3D,object`

The same as `points`, but for paths. You may use the same names as for points, `mfpic` knows in every situation, if the respective name is a point or a path.

`\objects{`⟨*list of objects*⟩`}`

The same as `\points`, but for objects and not necessarily connected to any environment.

5.5 making points and paths

`\point{`⟨*pointname*⟩`}`(*xvalue,yvalue*)
environments:`mfpic,object`

`\point{`⟨*pointname*⟩`}`(*xvalue,yvalue,zvalue*)
environments:`3D`

Both set the coordinates of a point which is declared under the logical name *pointname*. In the *xvalue,yvalue,zvalue* statements you can use parameters (but be sure that at the end there is no free parameter, better use the `rel` environment) and primitive operations of METAFONT, see below or manuals.

`\path{`⟨*pathname*⟩`}{`⟨*list of pointnames*⟩`}`
parameters:*connectsymbol*

pathname is a logical name, under which a path is declared. Likewise *list of pointnames* contains names of declared points. The parameter *connectsymbol* determines the mode of connecting the single points, default is connection by straight lines (you can set this by `\mplinear`). Other modes you get by

- `\mpquadratic` which is the same as `\mpcubic`, makes a BEZIER curve
- `\mptense`

  makes an almost straight curve, which is only rounded at the edges.
- `\mpbounded`

  makes a bounded curve.

If you are familiar with METAFONT you may change the mode of connection by manipulating `connectsymbol` directly.

`\cycle{`⟨*pathname*⟩`}{`⟨*list of pointnames*⟩`}`
parameters:*connectsymbol*

The same as `path`, but closes the path to a cycle.

`\circle{`⟨*pathname*⟩`}{`⟨*list of circle determining commands*⟩`}`
environments:`mfpic,object`

This makes a circular path (in the length independent coordinatesystem) by giving three points or the center and one point or the center and the radius. To this end use

`\threepoint{`⟨*3 points*⟩`}`

`\center{`⟨*1 point*⟩`}`

`\radius{`⟨*1 real number or parameter*⟩`}`

`\reverse`
reverses the direction of the curve.

`\arc{`⟨*pathname*⟩`}{`⟨*list of circular arc determining commands*⟩`}`
environments:`mfpic,object`

This makes a part of a circular path (in the length independent coordinatesystem) by giving three points or the center and one point and an angle or the center and the radius and one or two angles. To this end use

`\threepoint{`⟨*3 points*⟩`}`
the first point starts the arc, the second is the center of the respective circle and the arc stop at the direction of the third point (from the center of the arc)

`\center{`⟨*1 point*⟩`}`

`\radius{`⟨*1 real number or parameter*⟩`}`

`\arcangle{`⟨*angle in degrees*⟩`}`

`\startangle{`⟨*angle in degrees*⟩`}`
Gives the angle, at which (viewing from the center of the arc) the arc begins.

`\reverse`
reverses the direction of the curve.

`\ellipse{`⟨*pathname*⟩`}{`⟨*list of ellipse determining commands*⟩`}`
environments:`mfpic,object`

This makes an ellipse by giving enough parameters. To this end use

`\point{`⟨*point*⟩`}`
Determines one point through which the ellipse goes.

`\center{`⟨*1 point*⟩`}`

`\focuses{`⟨*pointa*⟩`}{`⟨*pointb*⟩`}`
determines the two focuses of the ellipse.

`\ratio{`⟨*1 real number or parameter*⟩`}`
Determines the ratio between the two halfaxis'.

`\halfaxis{`⟨*1 real number or parameter*⟩`}`
gives the big halfaxis.

`\angle{`⟨*angle in degrees*⟩`}`
gives the skew angle of the ellipse, if necessary, the default is 0.

`\reverse`
reverses the direction of the curve.

`\ellipticalarc{`⟨*pathname*⟩`}{`⟨*list of elliptical arc determining commands*⟩`}`
environments:`mfpic,object`
This (equivalently the `ellarc`) makes a part of an elliptical path by giving some parameters. To this end use

`\point{`⟨*point*⟩`}`
determines one point through which the full ellipse goes (not necessary the arc).

`\startpoint{`⟨*point*⟩`}`
determines the point where the ellipse starts.

`\center{`⟨*1 point*⟩`}`

`\focuses{`⟨*pointa*⟩`}{`⟨*pointb*⟩`}`
determines the two focuses of the ellipse.

`\ratio{`⟨*1 real number or parameter*⟩`}`
Determines the ratio between the two halfaxis'.

`\halfaxis{`⟨*1 real number or parameter*⟩`}`

`\angle{`⟨*angle in degrees*⟩`}`

`\arcangle{`⟨*angle in degrees*⟩`}`
Gives the length of the arc.

`\startangle{`⟨*angle in degrees*⟩`}`
Gives the angle, at which (viewing from the center of the ellipse) the arc begins.

`\startdirection{`⟨*point*⟩`}`
Gives the point, in direction of which (viewing from the center of the ellipse) the starting point of the arc lies.

`\reverse`
reverses the direction of the curve.

`\makefuncpath{`⟨*path name*⟩`}{`⟨*function*⟩`}`:*begin point*`--`*end point*
environments:`mfpic,object`
parameters:*connectsymbol*
Makes a path from the graph of *function* (x is the independent variable).

`\makeoneparapath{`⟨*path name*⟩`}{`⟨*x description*⟩`}{`⟨*y description*⟩`}`(*parameter*):*begin point*`--`*end point*

environments:`mfpic,object`

`\makeoneparapath{`⟨*path name*⟩`}{`⟨*x description*⟩`}{`⟨*y description*⟩`}{`⟨*z description*⟩`}`(*parameter*):*begin point*`--`*end point*

environments:`3D`

parameters:*connectsymbol*

The same, but work with an oneparametric description of the curve.

5.6 Transformations

You may want to change the actual transformation, e.g. to scaled or rotate or shift the output. Anything what follows works only in the `mfpic` and `object` environment.

`\trafo{`⟨*transformation*⟩`}`

Sets the actual transformation.

`\stoptrafo`

Makes the current transformation to be the identical transformation.

`\rotateby{`⟨*degrees*⟩`}`

Rotates around (0,0).

5.7 Drawing and Filling

To every macro listed below there are two more macros: one with *-form, which works instead of a path name with an explicite list of points representing the required path. The other macro has the same name and an ending `s`, this works with a list of path names. So there are besides `\drawpath` also `\drawpath*` and `\drawpaths`.

`\drawpath{`⟨*pathname*⟩`}`

parameters:*mplinethickness*

Draws the path given by *pathname* with a pen with thickness *mplinethickness*.

`\drawarrowpath{`⟨*pathname*⟩`}`

parameters:*arrowangle,arrowlength,arrowratio,mplinethickness,arrowobject*

The same a `drawpath`, but make an arrow at the endpoint in the ending direction.

`\drawdoublearrowpath{`⟨*pathname*⟩`}`

parameters:*arrowangle,arrowlength,arrowratio,mplinethickness,arrowobject*

The same a `drawarrowpath`, but puts an arrow at both endpoints.

`\drawdottedpath{`⟨*pathname*⟩`}[`⟨⟩`]`

environments:`object,mfpic,3D`

parameters:*dotpattern,mplinethickness*

Draws a dotted line along the given path, using the optional *dotpattern* and if this isn't valid, the actual `dotpattern`. The default of `dotpattern` is simply `10pt`, but you may change it to `2pt,1pt,3pt` or similiar (this means draw `2pt`, avoid `1pt`, draw `3pt`, avoid `2pt` and so on). It is also possible to use TEX characters for dotting, for a complete description we refer to the description of the parameter `dotpattern`.

`\overdrawpath{`⟨*pathname*⟩`}`

parameters:*overdrawratio,mplinethickness*

Draws the path given by *pathname* with a pen with thickness *mplinethickness*, but erases anything inside a bundle given by the ratio.

`\fillpath{`⟨*pathname*⟩`}`

Blackens the region inside the path.

`\filldrawpath{`⟨*pathname*⟩`}`

Blackens the region inside the path and on the boundary.

`\erasepath{`⟨*pathname*⟩`}`

Erases anything drawn and filled upto this moment under the path represented by *pathname*.

`\eraseinsidepath{`⟨*pathname*⟩`}`

Erases anything drawn and filled up to this moment under the region inside the path represented by *pathname*.

`\funcplot{`⟨*description of a function*⟩`}`:*xmin*`--`*xmax*

environments:`mfpic`

parameters:*funcplotthickness,functolerance*

Draws the graph of the function described (`x` is used as free variable) between *xmin* and *xmax* with thickness `funcplotthickness`. *functolerance* is a length which determines the distance between two values of the function to be computed. In *description of a function* the usual METAFONT syntax is used (see section below). WARNING:Be careful about not having additional blanks between the arguments!

`\onepara{`⟨*xparttext*⟩`}{`⟨*yparttext*⟩`}(`*parametername*`)`:*tmin*`--`*tmax*

environments:`object,mfpic`

parameters:*funcplotthickness,functolerance*

`\onepara{`⟨*xparttext*⟩`}{`⟨*yparttext*⟩`}{`⟨*zparttext*⟩`}(`*parametername*`)`:*tmin*`--`*tmax*

environments:`3D`

parameters:*funcplotthickness,functolerance*

The same as `funcplot`, but uses a one parametric description of a path. *parametername* gives the name of the parameter. WARNING:Be careful about not having additional blanks between the arguments!

`\twopara{`⟨*xparttext*⟩`}{`⟨*yparttext*⟩`}`:*umin*`--`*umax* `|`*ugrid*`|`*vmin*`--`*vmax*`|`*vgrid*`|`

environments:`mfpic,object`

`\twopara{`⟨*xparttext*⟩`}{`⟨*yparttext*⟩`}{`⟨*zparttext*⟩`}`:*umin*`--`*umax* `|`*ugrid*`|`*vmin*`--`*vmax*`|`*vgrid*`|`

environments:`3D`

parameters:*funcplotthickness,functolerance*

Draws a grid over the plain object given by the twoparametric functional, The names of the two parameters are `u` and `v`. The mesh is given by *ugrid* and *vgrid* which declare the number of curves to draw. WARNING:Be careful about not having additional blanks between the arguments!

`\getobject{`⟨*objectname*⟩`}`

environments:`object,mfpic,3D`

Includes all drawing and filling procedures from the named object.

5.8 Shading / Hatching

`mfpic` provides two kindes of shading: "shading" and "hatching". (My English isn't so good to make sure I'm using the names right.) The commands are available in all environments besides the `rel` environment. For both the shade and the hatch macro there is also a `*` version and a `s` version

`\shadepath`*pathname*(*shadedistance*,*shadethickness*, *shadeslant*)

Shades the inside of the path *pathname* by lines, which are *shadethickness* thin, separated by *shadedistance* and build an angle of *shadeslant* against the positive x-axis.

`\hatchpath{`⟨*pathname*⟩`}` `[`⟨*hatchobjectname,xhatchdistance,yhatchdistance,shadeslant*⟩`]`

parameters:*hatchobject,xhatchdistance,yhatchdistance,shadeslant*

This is more like PICTEX, it fills the region given by the path *pathname* with the optional *hatchobject*. If there is no hatchobject given, it uses that given by the actual *hatchobject* which is *squarehatch* unless changed by a new declaration of `hatchobject`. The parameters control the placement of the *hatchobject*, *shadeslant* is the angle between the x axis of the hatch grid (the x axis of the hatchobject is the same) and the x axis of the coordinate system.

`\betweenshade[`⟨*shadedistance,shadethickness,shadeslant*⟩`]`
`{`⟨*upper function*⟩`}{`⟨*lower function*⟩`}`:*xmin*`--`*xmax*

parameters:*connectsymbol*

Shades the region between the "lower" and the "upper" function (their graphs may cross, see `examp.tex` between the x-values *xmin* and *xmax*. WARNING:Be careful about not having additional blanks between the arguments!

`\betweenhatch[`⟨*hatchobject*⟩`]{`⟨*upper function*⟩`}{`⟨*lower function*⟩`}`:*xmin*`--`*xmax*

parameters:*connectsymbol,hatchobject*

Hatches the region between the lower and the upper function between the x-values *xmin* and *xmax*. WARNING:Be careful about not having additional blanks between the arguments!

5.9 Saving Pictures

`\savepicture{`⟨*picturename*⟩`}`

environments:`mfpic,3D`

Saves the current picture under the logical name *picturename*. The picture has all dimensions zero.

`\makepicture[`⟨*width,height,depth*⟩`]{`⟨*picturename*⟩`}{`⟨*objectname*⟩`}`

environments:`not necessary`

Makes a picture build from the object *objectname* available under *picturename*. You may specify by *width, height, depth* the dimensions of the picture.

`\getpicture{`⟨*picturename*⟩`}`

environments:`not necessary`

Insert the specified picture into the local context.

5.10 Labels

`\mpput{`⟨*label*⟩`}¡`*xshift,yshift*`¿[`⟨ *orientation*⟩`](`*point*`)`
environments:`mfpic,3D`

Puts *label* at the *point* with *orientation* and *xshift,yshift* are determined like in PICTEX, i.e. *orientation* gives the placement of the label. Possible are

- `[t]`
  with the top of the box at the point.
- `[b]`
  with the bottom at the point.
- `[B]`
  with the baseline at the point.
- `[l]`
  with the lefthand margin at the point.
- `[r]`
  with the righthand margin at the point.

Combinations are possible, default is the centered placement of the label at the point. The label is a horizontal box, if you need more than one line use `parbox` or something similiar from standard LATEX. *xshift,yshift* are two lengths giving some additional shift besides the values from *orientation*, which are also optional (cf. also PICTEX-manual). *point* is determined by a masked logical name or by explicite (in METAFONT-syntax) coordinates (without `(` and `)`). If you must use `(` and `)` in the syntax, try `\(` and `\)` again.

`\mpmultiput{`⟨*label*⟩`}<`*xshift,yshift*`>[`⟨ *orientation*⟩`]|{`⟨*list of points*⟩`}`
environments:`mfpic,3D`

The same as `\mpput`, but puts *label* on each of the determined points. Inside *list of points* there is available the macro

`\shifted`*shift vector*`\textit{number of points}`*starting point*

which stands for a list of equally distanced points. In the `3D` environment you must use `@` for specifying the shift vector and the starting point.

`\setput{`⟨*label*⟩`}¡`*xshift,yshift*`¿[`⟨ *orientation*⟩`](`*point*`)(`*point1*`)(`*point2*`)`
environments:`mfpic`
parameters:*overputsep*

Puts the label at the point, but at the same times determines *point1* to be be at the lower left corner of the label and *point2* at the upper right corner.

`\overput{`⟨*label*⟩`}¡`*xshift,yshift*`¿[`⟨ *orientation*⟩`](`*point*`)`
environments:`mfpic,3D`
parameters:*overputsep*

The same as above, but erases anything which was drawn upto this moment and is placed under the box build by *label*. This box is thereby enlarged by *overputsep* on every side.

`\multioverput{`⟨*label*⟩`}<`*xshift,yshift*`>[`⟨ *orientation*⟩`]{`⟨*list of points*⟩`}`
environments:`mfpic,3D`
parameters:*overputsep*

Conglomerate of `overput` and `mpmultiput`.

`\mpangleput[`⟨*minimal radius*⟩`]{`⟨*label*⟩`}{`⟨*point1*⟩`} {`⟨*point2*⟩`}{`⟨*point3*⟩`}`
parameters:*overputsep,mplinethickness,minangleradius*
environments:`mfpic,3D`

Puts *label* into a part of a circle (with *minimal radius*), the center is given by *point2*, the fronting arc goes from the direction of *point1* to the direction of *point3*.

`\mpperp[`⟨*radius*⟩`]{`⟨*center*⟩`}{`⟨*direction point*⟩`}`
parameters:*mplinethickness,dotratio,dotthickness*
environments:`mfpic`

`\mpperp[`⟨*radius*⟩`]{`⟨*point1*⟩`}{`⟨*point2*⟩`}{`⟨*point3*⟩`}`
parameters:*mplinethickness,dotratio,dotthickness*
environments:3D

Puts a dot with thickness `dotthickness` inside a quartercircle whith *center* or *point2* and beginning direction specified by *direction point* respectively *point1* (both are in METAFONT syntax, with masks). The quartercircle is positively oriented, the distance of the dot from the *center* is *dotratio* times *radius*.

5.11 The Coordinate system

`\bounds`*xmin*`--`*xmax ymin*`--`*ymax*
environments:`mfpic`

`\bounds`*xmin*`--`*xmax*`|`*ymin*`--`*ymax*`|`*zmin*`--`*zmax* `|`
environments:3D

In both cases you change the bounds for the picture. This commands effects the appearance of the picture, as the last valid values for `\xmin,\xmax,\ymin,\ymax,\zmin,\zmax` give `mfpic` that values, to place the picture into the document. The bounds are also relevant for clipping and dividing the picture.

`\xaxis`(*yval*)
environments:`mfpic`

`\xaxis`(*yval,zval*)
environments:3D
parameters:*arrowlength,arrowangle,arrowratio,mplinethickness*

Draws an arrowed coordinate axis in x-direction, shifted to the value `y=\it yval`.
Similiarly you have `\yaxis,\zaxis`

`\axises`(*xval,yval*)
environments:`mfpic`

`\axises`(*xval,yval,zval*)
environments:3D
parameters:*arrowlength,arrowangle,arrowratio,mplinethickness*

Draws all coordinates at onces.

`\xticks{`⟨*list of tick commands*⟩`}`
parameters:*pointstring,mantisse,overputsep,shorttickratio,ticklength*
environments:`mfpic,3D`

Draws the ticks specified by the commands. As commands are possible:

`\tickssshort,tickslong`
Gives short or long ticks.

`\numbered[`⟨*length of mantisse*⟩`]`
numbers the ticks.

`\labeled{`⟨*list of labels*⟩`}`
Gives the ticks the specified labels.

`\numfunc{`⟨*description of function*⟩`}`
Gives the place of the tick, i.e. the tick to number x is to be placed at $function(x)$.

`\logged`
Makes a logarithmic scale.

`\labelstyle`
Gives the style in which to set the labels. It takes one argument containing the label, i.e. a correct way to redefine it is `\labelstyle#1{$#1$}` . There is no need that uch a declaration occurs inside `\xticks` .

`\range`*begin value*`:`*step value*`:`*end value*
Gives the range in which the ticks have to be set, approximately distanced by *step value*.

`\mesh`*begin value*`--`*end value*`|`*number of intervalls*`|`
Between *begin value* and *end value* there are placed *number of intervalls + 1* ticks equidistantly.

`\at{`⟨*list of numbers*⟩`}`
Sets ticks at each place corresponding to one number in the list, numbers can be given in METAFONT syntax.

`\shifted{`⟨*shift value*⟩`}`
Shifts the ticks to the given value, only in the `mfpic` environment.

`\shifted`(*shift yvalue,shift zvalue*)
Shifts the ticks to the given value, only in the `3D` environment.

`\left,\up,\right,\down,\centered`
Determines the placement of the tick.

`\make`
Realizes the commands given up to this moment.

`\yticks,\zticks`
The same as above, but with the y and the z axis.

`\mpgrid`*xtimesytimes*
environments:`mfpic`
Makes a grid within the actual bounds. (the times give the number of lines.)

`\clip`
environments:`mfpic,3D`
Clips the current picture, i.e. no ink is given points, which lie outside the current bounds of the picture. (In the environment `3D` these are all points, the picture of which is situated inside the picture of the bounding cube.

`\accountingon`
Makes any `\bounds` and `\mpput,\mpmultiput,\overput` command contribute to the bounding dimensions of the actual `mfpic` or `3D` environment.

`\accountingoff`
Similiar.

`\dividechar`*xtimes*x*ytimes*
If your pictures are too detailed, there would be the necessity to split it into subpictures, which does this macro. *xtimes* and *ytimes* are the division rates.

`\kavalierperspective`,`parallelperspective`
environments:3D
parameters:*xkavalier*,*ykavalier*
Choose the perspective used to show the 3D–things. In general (if you have some curved lines like circles) use `parallelperspective`. There is a default value, but this is a strange transformation.

`\kavalier`(*xkavalier*,*ykavalier*)
environments:3D
Sets the parameters *xkavalier* and *ykavalier*, default is `kavalier(0.5cos 45°,0.5sin 45°)`

`\viewperspective`
environments:3D
parameters:*view*
Gives another possible perspective, which arises, if one looks at the 3D picture from a direction specified by the azimutal and horizontal angle in *view*.

`\view{`⟨*azimutal angle*⟩`}{`⟨*horizontal angle*⟩`}`
Specifies the angles for use in `viewperspective`. The default is set to `view{`⟨*30*⟩`}{`⟨*-37.5*⟩`}`.

5.12 Parameters

Parameters are values which are important for some METAFONT macros in `mfpic`. Suppose you are given a parameter named `foo`. If you declare this name to be a parameter, there are two macros: `foo` and `\@foo\verb`. The latter stores the value of the parameter, the former allows you to give this parameter a new value. Inside METAFONT text you refer to this parameter by `foo` only.

Parameter–making and changing is not restricted to any environment (but depends on grouping).

You have the possibility to declare some parameters.

`\parameterdef{`⟨*parametername*⟩`}`

`\parameterdefs{`⟨*list of parameternames*⟩`}`
Declares the above macros. There is no initial value!

`\pairparameterdef{`⟨*parametername*⟩`}`

`\pairparameterdefs{`⟨*list of parameternames*⟩`}`
Declares the above macros to stand for twodimensional parameters. There is no initial value!

`\lengthdef`*lengthname*

`\lengthdefs`*list of lengthnames*
The same as above, but it is assumed, that the parameter declares some length. To set this parameter, both `\foo{10pt}` and `\foo10pt` are allowed (also `\foo{anotherfoo}` if `another foo` is another parameter), but don't use the `setlength` from LaTeX!

`\getparameters{`⟨*list of parameters*⟩`}`
Gives METAFONT the currentvalue of the parameters in *list of parameters*.

`\forparameter{`⟨*parametername*⟩`}` `{`⟨*list of values*⟩`}{`⟨*commands*⟩`}`
Executes the commands for several values of one parameter. Inside the environments one can even use the syntax from the `\begin{rel}` environment, see also the macro `\real` below.

`\forparameters`(list of parameters) `{`⟨*list of value-vectors*⟩`}{`⟨*commands*⟩`}`
The same, but with several parameters. The list is build from entries of the kind (value1,value2,...).

`\real{`⟨*value in coordinates*⟩`}`
environments:`object,2D,3D`
Inside the `\forparameter`, it convertes a coordinate-description into real parameters

5.13 In Cases of Emergency, Basic Control

`\onlyputs`
Only put commands are processed. This command should be used if it is not necessary to execute draw, shade, fill and so on commands.

`\putsanddraws`
Standard, anything is executed.

`\optimize`
Optimizes the actual picture a the present state (this can help to deal with memory overflow).

`\mfcmd{`⟨*command text*⟩`}`
This macro writes the *command text* directly to the METAFONT file, using a TEX `write` command. This can have some rather bizarre consequences, so using it is not recommended unless you know, what you do. (Don't change even the `mpjobname.log` file by show commands! If this happens, delete the `.log` afterwards).

`\mfdrawcmd{`⟨*command text*⟩`}`
The same, but depending whether `\onlyputs` or `\putsanddraws` are active, the command text is written into the `.mf` file.

6 List of current parameters

`arrowangle, arrowratio, arrowlength`
Three parameters for use in drawing arrows. defaults: 25, 1, 10pt

`xmin, ymin, zmin, xmax, ymax, zmax`
Give the bounds, default:0.

`funcplotthickness, functolerance`
The thickness for drawing function graphs and the steplength taken to get the graph good enough. Default values 0.4pt and 10pt.

`mplinethickness`
Thickness of normal lines, default 0.4pt.

`dotpattern`

This is much more complex. In general, `dotpattern` is a list and each entry is one of the following

- `a length` A piece of the given length is drawn (or not drawn, if the number of the piece is even) with the actual pen.
- `\dotsymbol{`⟨*{⟨symbol text⟩}[⟨symbol placement⟩]*⟩`}`

  Instead of drawing a symbol is put into the middle of the specified piece (not depending on the number of the piece, according to the placement specifiers, see the macros `\mpput,\mpmultiput`). To change the original measures of the symbol just follow the command (before the next colon) by another length which replaces the standard $\sqrt{h^2+w^2}$, h and w being height and width.
- `\dotdirsymbol{`⟨*symbol text*⟩`}`

  The same, but the standard length is not $\sqrt{h^2+w^2}$ but depends from the direction. For changing the immanent measures of the symbol just specify another height and width by a following `(h\string\,w)`.
- `\dotobject{`⟨*{⟨objectname⟩}length,*⟩`}`
- `\dotdirobject{`⟨*{⟨objectname⟩}(height width),*⟩`}`

  The same, but uses an object. You must specify some measures!
- `parameters`

  Every entry can begin with some parameter settings which are valid from this moment.

The default dotpattern is simply `10pt`, which means that the path is divided into pieces of length 10pt which are drawn and not drawn consequitively.

`shadeslant`

Given in degrees, it determines the angle, by which you have to rotate the shading lines or the hatchobject inside shading or hatching. Its default value is 0.

`shadedistance`

Determines the distance between the shading lines for shading. Its default value is 10pt.

`shadethickness`

Determines the thickness of the shading lines for shading. Its default value is 0.4pt.

`xhatchdistance, yhatchdistance`

Distances, by which the hatchobjects have to be separated. Default 1pt.

`hatchlength`

Parameter to determine the length of little symbols (dots) in the `\squarehatch` and `\circlehatch` object. Default 0.25pt.

`hatchthickness`

The thickness with which the draw commands in hatchobjects are working. Default 0.2pt.

`hatchobject`
The actual default hatchobject, default is squarehatch. From the beginning there are available the following hatchobjects:

`xhatch`

`crosshatch`

`fullcirclehatch`

`circlehatch`

`squarehatch`

These can be made default by the macro with the same name. But you may also specify another hatchobject of your choice, if you have built one.

`arrowobject`
This gives the object which is drawn if an arrow should be drawn, e.g. for making some axises. There are two objects built in:

`fullarrow`

`openarrow`

Both can be activated by `fullarrows` or `openarrows` but the user is free to use any other object defined by him.

`objecttransform`
Gives the transformation, which applies to the objects included afterwards.

`xkavalier,ykavalier`
Set the ratios for the z-axis in both the kavalier and the parallel perspective, i.e. the point $(0, 0, 1)$ is mapped onto $(xkavalier, ykavalier)$.

`overputsep`
Gives the amount of space added to a label on each side, if the part of the picture under the label is erased by an `overput` command. Also used for the additional distance between ticks and labels. Default: 1pt.

`pointsymbol`
Symbol between integer and rational part of labels for ticks, default is `..`

`mantisse`
Default value of length of mantissa by labeling ticks, originally set to 1.

`ticklength`
Length of long ticks, default 10pt.

`shorttickratio`
Ratio by which the tick is shortened in short ticks, default 1/2.

`dotratio`
Ratio between radius of the quartercircle and the distance of the dot from the center of the quartercircle in `mpperp`, default 1/2.

`dotthickness`
Thickness of the dot in the same macro, default 2pt.

`minangleradius`
Minimal radius of arcs used by `\mpangleput` and `\mpperp`.

7 METAFONT syntax, which can be freely used

Inside the `\begin{rel}\end{rel}` environment (and the arguments of the commands `\point,\mpput,\mpmultiput,\overput` and the stared drawing macros you can use all the syntax, METAFONT allows for simple equations or declarations. E.g. the `=` denote equality, `:=` an assignment. All names used denote real quantities, unless stated otherwise (do declare a 2dimensional vector `zz` use `pair zz;`). All statements have to be separated by semicolons. A short description for the point dividing the line between the points (or numbers) `z1` and `z2` in the ratio `q:(1-q)` is `q[z1,z2]` (it is also possible to have $q < 0$ or $q > 1$). METAFONT allows the parameter `whatever` to denote an arbitrary real quantity. So `whatever[z1,z2]` is an arbitrary point on the straight line through `z1` and `z2`. There are the following functions available: `sin,cos,tan,exp,log,abs` (also for complex numbers = plain vectors) `sqrt,*,**,/,+,-`. METAFONT gives you also `sind,cosd`, which work with arguments in degrees, and `mexp` and `mlog`, given by $\mathrm{mlog}(x) = 256 * \log(x)$ and $\mathrm{mexp}(x) = 256 * \exp(x/256)$. Also, grouping is made by `(` and `)`, inside the put commands and point commands `\(` and `\)` are required. You can use this syntax wherever parametervalues are required, but it doesn't work with lengths. For a deeper description of the METAFONT syntax we refer to the METAFONT book by Donald E. Knuth.

Warning: METAFONT has a largest value of 4096, thus you should be sure, that your expressions lead not to such large numbers.

8 Further Plans

It would be preferable, if there is a plain TEX version, but it would be to puzzling in the moment to copy the respective hacks from the `latex.tex` file into some file named `mfpic.tex` to be inputted into TEX. It is also nice to have additionally an `.aux` file, such that there is no problem in finishing the output. But this is not a big problem besides time.

Further, it would be good to expand the 3D–part, especially introducing 3D–objects.

It seems to be possible to extend shading even to the case, where the user gives a TEX box, the material inside which should fill the region. But at the moment I think a usergiven object would be enough.

9 Communication

Because of lack of time, the author couldn't test all macros as deep as he liked to. Thus he would like to hear, how this styles works and what has to be made better. Please contact to Volkmar Liebscher (`liebsche@gsf.de`)

10 Acknowledgement

I like to thank D. Kaiser for some advisement in the algorithm for drawing dotted curved lines. One further thank yields M. Malarski for finding the other `mfpic` project at network. I'm very grateful towards F. Ditrich, Th. Fischer, G. Richter,

G. Schmidl, N. Zeh and K.Basler, who standed the uncountably many errors in former versions of this style.