minim-mp

version 2025/1.7

author Esger Renkema
contact minim@elrenkema.nl

This package provides low-level mplib integration for plain lua(la)tex. In order to use it, simply say `\input minim-mp.tex` (from plain tex) or `\requirepackage {minim-mp}` from latex. After this, `\directmetapost [ options ] { mp code }` will result in a series of images corresponding to the `beginfig() ... endfig;` statements in your `mp code`. Every image will be in a box of its own.

This package can also be used as a stand-alone metapost compiler. Saying

```
luatex  --fmt=minim-mp  your_file.mp
```

will create a pdf file of all images in `your_file.mp`, in order, with page sizes adjusted to image dimensions. You might need generate the format first, this is done with

```
luatex  --ini  minim-mp.ini
```

Use minim-lamp instead of minim-mp for a latex-based version of the minim-mp format. With minim-lamp, for specifying the contents of the preamble, you can use `verbatimtex ... etex;` statements at the top of your file. Concluding the preamble with `\begin{document}` is optional, as both `\begin` and `\end{document}` will be inserted automatically if omitted.

TeX-MetaPost compatibility notes

All metapost code to-be-run will be read under a special catcode regime in which most 'special' characters (including `#` and `%`) are inert. Any control sequences will be expanded fully, however, as inside an `\edef`. Keep this in mind and define any macros you want to use in a `btex ... etex;` environment as `\protected`. With the latex package, latex's 'protected' macros are safe-to-use, too.

Every call to `\directmetapost` opens and closes a separate metapost instance. If you want your second call to remember the first, you will have to define a persistent metapost instance. This will also give you more control over image extraction. See below under „Metapost instances". The `options` will also be explained there (for simple cases, you will not need them).

LaTeX compatibility notes

The latex package minim-mp.sty contains the full plain interface, but also provides a proper `metapost` environment as an alternative to `\directmetapost`. The `metapost` environment has no persistent backing instance, but you can create a similar environment `envname` that does with `\newmetapostenvironment [options] {envname}`.

Inside the above environments, you can use `\mpcolor {name}` to insert the proper colour values. Unlike in luamplib, this will work in any context where metapost expects a colour. An `\mpdim` macro is present too, but quite superfluous since the contents of the metapost environments are fully expanded before metapost sees them: this means you can use `\the` and `\the\dimexpr` like everywhere else. Tex's registers can also be accessed from within metapost itself (see below).

When the package is loaded with the option `luamplib`, minim-mp will try and act as a drop-in replacement for luamplib. The effort made is not very great, but it will define an `mplibcode` environment, as well as the `\mplibcodeinherit`, `\mplibshowlog`, `\mplibsetformat` and `\mplibnumbersystem` switches; also the macros `\everymplib` and `\everyendmplib`.

MetaFun compatibility notes

The metafun format is supported to the extent that it should not immediately crash. Otherwise, useing metafun is neither supported nor recommended.

Metapost extensions

You can set the baseline of an image with `baseline(p)`. There, `p` must either be a point through which the baseline should pass, or a number (where an x coordinate of zero will be added). Transformations will be taken into account, hence the specification of two coordinates. The last given baseline will be used.

You can write to tex's log directly with `texmessage "hello";`. You can feed it a comma-separated list of strings and numbers, which will be passed through `string.format()` first.

You can write direct pdf statements with `special "pdf: statements"` and you can add comments to the pdf file with `special "pdfcomment: comments"`. Say `special "latelua: lua code"` to insert a `late_lua` whatsit. All three specials can also be given as pre- or postscripts to another object. In that case, they will be added before or after the object they are attached to. Do note that all `special` statements will appear at the beginning of the image; use pre- and postscripts to drawing statements if the order matters.

Partial paths and the even-odd rule

While metapost fills paths according to the winding number, the pdf format also supports filling according to the even-odd rule. You can use this method with the `eofill` and `eofilldraw` drawing statements. Multiple contours can be filled at once if all but the last are drawn with `nofill`. The latter is especially useful for cutting something out of a shape, as it saves you the hassle of stitching the paths together.

The macros `multi(draw|fill|filldraw|eofill|eofilldraw)` take a list of paths between parentheses and can be followed by the usual drawing options. For example, `multidraw (contours "example" of "tenbf") withpen currentpen scaled 1/10;` will give the word example in a thin outline.

The clipping shorthands `clipout` and `clipto` take a list of paths as a 'text' parameter and either clip their ensemble out of the current picture, or the current picture to the ensemble. You will have to ensure yourself that the path has the right orientation for clipping (i.e. you may have to insert the proper `reverse` operator yourself).

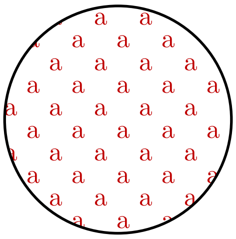
Tiling patterns

The specification `withpattern(<name>)` added to a `fill` statement will fill the path with a pattern instead of a solid colour. If the pattern itself does not contain any colour statements, it will have the colour given by `withcolor`. Stroking operations (the `draw` part) will not be affected. Patterns will always look the same, irrespective of any transformations you apply to the picture.

To define a pattern, sketch it between `beginpattern(<name>) ... endpattern(xstep, ystep);` where `<name>` is a suffix and `(xstep, ystep)` are the horizontal and vertical distances between applications of the pattern. Inside the definition, you can draw the pattern using whatever coordinates you like; assign a value to the `matrix` transformation to specify how the pattern should be projected onto the page. This `matrix` will also be applied to `xstep` and `ystep`.

You can also change the internal variable `tilingtype` and the normal variable `painttype`, although the latter will be set to 1 automatically if you use any colour inside the pattern definition. Consult the pdf specification for more information on these parameters.

You can use text inside patterns, as in this example:



```
% define the pattern
picture letter; letter = maketext("a");
beginpattern(a)
    draw letter rotated 45;
    matrix = identity rotated 45;
endpattern(12pt,12pt);
% use the pattern
beginfig(1)
    fill fullcircle scaled 3cm withpattern(a) withcolor 3/4red;
    draw fullcircle scaled 3cm withpen pencircle scaled 1;
endfig;
```

A small pattern library is available in the `minim-hatching.mp` file; see the accompanying documentation sheet for an overview of patterns.

Box resources

Box resources (XForms) saved by tex can be included with `boxresource nr`. (Note that `nr` is not a box number, but the index returned by `\lastsavedboxresourceindex`.) The result will be an image object with the proper dimensions. This image can be transformed in any way you like, but you cannot inspect the contents of the resource within metapost.

Box resources can also be created from within metapost: `<id> = saveboxresource (<attributes>) <picture>` returns a number identifying the resource. The `<attributes>` can be given in the same way as the arguments to `setgstate` (see below). XForms defined through metapost are available to other metapost instances but not to tex; they are drawn with the same command (`boxresource nr`). There remains a subtle difference, however: metapost-defined box resources are placed at their original origin, tex-defined resources at their lower left corner.

Advanced PDF graphics

You can use `savegstate` and `restoregstate` for inserting the `q` and `Q` operators; these must always be paired, or horrible errors will occur. You may need them if you use `setgstate(<params>)` for modifying the extended graphical state (ExtGState). The `params` must be a comma-separated `Key=value` list, with all `values` being suffixes. The latter restriction may require the use of additional variables, but as this is a very low-level command, it is best to wrap it in a more specialised macro anyway. The `withgstate (<params>)` can be added to a drawing statement and includes saving/restoring the graphical state.

Note that while you could try and use `setgstate` for modifying variables like the line cap or dash pattern, the result of doing so will be unpredictable, since such changes will be invisible to metapost. Its intended use is restricted to graphics parameters outside metapost's scope.

Transparency

For applying transparency, `setalpha(a)` updates the `CA` and `ca` parameters as a stand-alone command and `withalpha(a)` can be used in a drawing statement where it will save/restore the graphical state around it. For applying transparency to an ensemble of drawing statements, `transparent (a) <picture>` instead of `draw <picture>` will create and insert the proper transparency group. The transparency group attributes can be set with the string internal `transparency_group_attrs`, while for all three macros the blend mode can be set with the string internal `blend_mode` (it will be added whenever set).

Additions to plain.mp

Minim-mp provides a few elementary macros and constants that are conspicuously absent from plain.mp; I hope their addition is uncontroversial. All are provided in the file minim.mp. This file is safe for use with other metapost processors.

The constants are `pi` (355/113), `fullsquare`, `unitcircle` and the cmyk-colours `cyan`, `magenta`, `yellow` and `key`. The macros are `clockwise`, `xshifted`, `yshifted`, `hflip` and `vflip`, where the flips are defined in such a way that `p & hflip p` gives the expected result.

Version 1.2 brought the following additions: `save_pair`, `save_path` etc. etc. that save and declare in one go; the missing trigonometric functions `tand`, `arcsind`, `arccosd` and `arctand`, and the unit circle segment drawing function arc(θ_0,θ_ℓ) (taking a starting angle and arc length, both in degrees).

Version 1.6 brought `empty` for testing empty suffixes, `typeof(v)` expanding to the type (`numeric`, `pair` etc.) of variable `v`, `getbounds` as complement to `setbounds`, the (twodimensional) exterior product `p extprod q` and the test `collinear(p,q,r)`.

Running tex code

You can include tex snippets with either `maketext "tex text"` or `btex ... etex` statements. The tex code will be executed in the current environment without an extra grouping level. The result of either statement at the place where it is invoked is an image object of the proper dimensions that can be moved, scaled, rotated and mirrored. You can even specify a colour. Its contents, however, will only be added afterwards and are invisible to metapost.

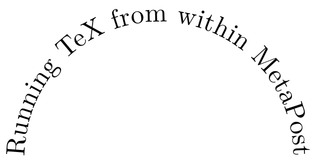
Arbitrary tex statements may be included in `verbatimtex ... etex`, which may occur anywhere. These `btex` and `verbatimtex` statements are executed in the order they are given.

When including tex code in metapost, do keep in mind that every metapost snippet read by `\directmetapost` or `\begin{metapost}` will be expanded fully (as inside an `\edef`) before execution. This may expand your macros too early! The best solution is to define your own macros as `\protected` or using latex's more convoluted protection equivalent, although a well-placed `\noexpand` or, as a last resort, wrapping the entire tex fragment in `\unexpanded{...}` will work as well.

You can also use metapost's `infont` operator, which restricts the text to-be-typeset to a single font, but returns an `picture` containing a `picture` for each character. The right-hand argument of `infont` should either be a (numerical) font id or the (cs)name of a font (without backslash). A traditional font name (e.g. ‘cmtt10’) is also accepted; this will silently load the font.

One possible use of the `infont` operator is setting text along curves:

```
beginfig(1)
    save t, w, r, a; picture t;
    t = "Running TeX from within MetaPost" infont "tenrm";
    w = xpart lrcorner t = 3.141593 r;
    for c within t :
        x := xpart (llcorner c + lrcorner c)/2;
        a := 90 - 180 x/w;
        draw c rotatedaround((x,0), a)
               shifted (-r*sind(a)-x, r*cosd(a));
    endfor
endfig;
```



By default, the `maketext` operator is used for typesetting labels. You can, however, order de `label` macro to use `infont` instead by setting `maketextlabels` to `false`.

Access to font contours

Both the `maketext` and `infont` operators return pictures that are opaque to metapost. For accessing the contours that make up the characters in a string, you will need the `glyph g of f` operator. It is a bit more versatile than its traditional metapost counterpart: `g` may also be the name of the glyph instead of its index, while `f` can either be a font id or font name (as with `infont`).

A variant of `glyph of` is the `contours s of f` macro, which first typesets the string `s` in the same way as `infont` does (so that kerning and font shaping are applied), but returns a (comma-separated) list of contours. This list is fit for use in a for loop. Note that due to rounding errors, the glyph contours will not match the output of `infont` exactly.

Be also aware that the contours returned by these operators may be disjoint: a letter `o`, for example, will consists of two paths. This means you cannot recreate the characters by just filling each contour, as that would turn the `o` into a filled-in circle. Instead, you must use `multifill` on the entire output of `glyph of` or `contours of` (see above under „Partial paths”).

Running lua code

You can call out to lua with `runscript "lua code"`. For this purpose, each metapost instance carries around its own lua environment so that assignments you make are local to the instance. (The global environment can still be accessed behind the `_G` variable.)

Every runscript environment is augmented with a small set of useful functions; this set can be found in the `M.mp_functions` table (with `M = require 'minim-mp'`; see below under „Lua interface”).

When using `runscript`, you must ensure its argument is a correct lua program. As an escape hatch, raw strings can be passed to lua with `runscript` `("[[function_name]]" & raw_string)`. This will return the result of the function `function_name` applied to `raw_string` as a lua string.

Returning values from lua

If your lua snippet returns nothing, the `runscript` call will be invisible to metapost. If on the other hand it does return a value, that value will have to be translated to something metapost can understand. You can return a point, colour or transform by returning an array of two to six elements (excluding five). Numbers and strings will be returned as they are. For other return values, `tostring()` will be called.

Note that lua strings will be returned as metapost code, not as metapost string literals! If you want to return the latter, you must quote the string yourself; the runscript environment contains a `quote(s)` function for this purpose, which also escapes all double quotes in the string `s`. When returning dimensions, you must be wary of arithmetic overflows: every useful dimension expressed in scaled points will be too large for metapost. Use `sp_to_pt(nr)` instead, which converts dimensions to points.

Passing values to lua

Do keep in mind that metapost and lua represent numbers in different ways and that rounding errors may occur. For instance, metapost's `decimal epsilon` returns `0.00002`, which metapost understands as `1/65536`, but lua as `1/50000`. Use the metapost macro `hexadecimal` instead of `decimal` for passing unambiguous numbers to lua.

Additionally, you should be aware that metapost uses slightly bigger points than tex, so that `epsilon` when taken as a dimension is not quite equal to `1sp`. Use the metapost macro `scaledpoints` for passing to lua a metapost dimension as an integral number of scaled points.

Strings can be passed to lua with the `lua_string` macro, which escapes the necessary characters and then surrounds its argument with quotes. A generic macro for passing values to lua, finally, is `quote_for_lua`, which automatically converts strings, numbers, points and colours to (metapost) strings that lua can understand.

Querying tex and lua variables

Stitching together lua snippets by hand is not very convenient. Therefore, minim-mp provides three helper macros that should cover most lua interaction. For running a single lua function, `luafunction <suffix> (<arguments>)` returns the result of the function `str <suffix>` applied to any number of arguments, which are quoted automatically. Variables can be queried with `luavariable <suffix>` and set with `setluavariable <suffix> = <value>;`.

The details of metapost tokenisation make these macros rather powerful: you can not only say e.g. `luavariable tex.jobname` to get the current jobname, but even define a `texvariable` macro with

```
vardef texvariable @# = luavariable tex @# enddef;
```

and have `texvariable jobname` work as expected.

For accessing count, dimen, attribute or toks registers, the macros are `tex.count [number]` or `tex.count.name` [etc. etc.] for getting and `set tex.count [number] = value` or `set tex.count.name = value` etc. for setting values.

Metapost instances

You can define a new persistent metapost instance with `\newmetapostinstance [ options ] \id`. An instance can be closed with `\closemetapostinstance \id`. The `options` are directly passed to lua, as a table, and can contain the following keys:

Option	Default	Description
`jobname`	`':metapost:'`	Used in error messages.
`format`	`'plain.mp'`	Format to initialise the instance with.
`mathmode`	`'scaled'`	One of `scaled`, `decimal` or `double`.
`seed`	`nil`	Random seed for this instance.
`catcodes`	(internal value)	Catcode table for `btex ... etex`.
`env`	copy of `_G`	Lua environment for `runscript`.

See below under „Lua interface” for greater control over these parameters.

Retrieving images from instances

Now that you have your own instance, you can run chunks of metapost code in it with `\runmetapost \id { code }`. Any images that your code may have contained will have to be extracted explicitly. This is possible in a number of ways, although each image can be retrieved only once.

All metapost images have an `\hbox` as outermost container. Do not unpack this box: doing so will cause its dimensions to be lost.

`\getnextmpimage \id` – Writes the first unretrieved image to the current node list. There, the image will be contained in a single box node.

`\getnamedmpimage \id {name}` – Retrieves an image by name and writes it to the current node list.

`\boxnextmpimage \id box-nr` – Puts the next unretrieved image in box `box-nr`. The number may be anything tex can parse as a number.

`\boxnamedmpimage \id box-nr {name}` – Puts the image named `name` in box `box-nr`.

Say `\remainingmpimages \id` for the number of images not yet retrieved and `\getallmpimages \id` to insert all remaining images.

Finally, as a shorthand, `\runmetapostimage \id { code }` will add `beginfig ... endfig;` to your `code` and write the resulting image immediately to the current list.

Associated token lists

Associated to every instance are four token registers containing tex and/or metapost code to be inserted before or after each executed chunk. You can access them with `\everymp \id {pre|post} {tex|mp}` (e.g. `\everymp1{pre}{mp}`). The token lists where `id` = 0 do not correspond to any instance and will apply to every instance instead. All four token lists are empty by default if you use minim-mp with plain tex. The latex package, on the other hand, uses them for its own purposes, so take care in that case to use `\tokspre` and `\toksapp` instead of overwriting their contents. As a bonus, when using the latex package, you can use the environment name instead of the numerical `\id`.

Lua interface

In what follows, you should assume `M` to be the result of

```
M = require('minim-mp')
```

as this package does not claim a table in the global environment for itself.

You can open a new instance with `nr = M.open {options}`. This returns an index in the `M.instances` table. Run code with `M.run (nr, code)` and close the instance with `M.close (nr)`. Images can be retrieved only with `box_node = M.get_image(nr, [name])`; omit the `name` to select the first image. Say `nr_remaining = M.left(nr)` for the number of remaining images.

Each metapost instance is a table containing the following entries:

`jobname`	The jobname.
`instance`	The primitive metapost instance.
`results`	A linked list of unretrieved images.
`status`	The last exit status (will never decrease).
`catcodes`	Number of the catcode table used with `btex ... etex`.
`env`	The lua environment for `runscript`.

Default values for the format and number system are available in the `M.default_format` and `M.default_mathmode` variables. The full initialisation sequence for new instances (i.e. the first metapost chunk, that is executed automatically) is contained in the `M.init_code` table.

The table passed to `env` defaults to a copy of the global table `_G`, and will be augmented with the contents of `M.mp_functions`. You can add your own functions to this table, if you wish.

Adding new metapost specials

You can extend this package by adding new metapost specials. Specials should have the form `"identifier: instructions"` and can be added as pre- or postscript to metapost objects. A single object can carry multiple specials and a `special "..."` statement is equivalent to an empty object with a single prefix.

Handling of specials is specified in three lua tables: `M.specials`, `M.prescripts` and `M.postscripts`. The `identifier` above should equal the key of an entry in the relevant table, while the value of an entry in one of these tables should be a function with three parameters: the internal image processor state, the `instructions` from above and the metapost object itself.

If the `identifier` of a prescript is present in the first table, the corresponding function will replace normal object processing. Only one prescript may match with this table. Functions in the the other two tables will run before or after normal processing.

Specials can store information in the `user` table of the picture that is being processed; this information is still available inside the `finish_mpfigure` callback that is executed just before the processed image is surrounded by properly-dimensioned boxes. If a `user.save_fn` function is defined, it will replace the normal saving of the image to the image list and the image node list will be flushed.

Logs and debugging

Metapost log output of every processed chunk of metapost code will be printed to the main log file in a quoted form. The logs will be duplicated to the terminal if an error occurs or if the variable `M.on_line` is `true` (the default is `false`, but set to `true` in the stand-alone format files). Because a single error message might trigger many others, log output to the terminal (but not the log file) will be cut after a certain number of errors: this behaviour is governed by the `M.max_errprint` variable (default `1`; set to `0` to disable).

You can enable (global) debugging by saying `debug_pdf` to metapost, `\debugmetapost` to tex or `M.enable_debugging()` to lua. This will write out a summary of metapost object information to the pdf file, just above the pdf instructions that object was translated into. For this purpose, the pdf will be generated uncompressed. Additionally, the metapost code itself will be echoed to the log, as well as `runscript` and `maketext` invocations, and a small summary of every generated image will be written to log and terminal.

For debugging the results of `maketext` and the like, you can set the `debug_tex_bboxes` variable to `true` and have their resulting bounding boxes drawn on the pdf.

Licence

This package may be distributed under the terms of the European Union Public Licence (EUPL) version 1.2 or later. An english version of this licence has been included as an attachment to this file; copies in other languages can be obtained at

https://joinup.ec.europa.eu/collection/eupl/eupl-text-eupl-12