minim

version 2024/1.5

author Esger Renkema
contact minim@elrenkema.nl

This is a modern plain format for the LuaTeX engine, adding improved low-level support for many LuaTeX extensions and newer PDF features. While it can be used as drop-in replacement for plain TeX, it probably is most useful as a basis for your own formats.

Most features included in the format are provided by separate packages that can be used on their own; see the packages

minim-mp	for mplib (MetaPost) support
minim-math	for Unicode mathematics
minim-pdf	for hypertext and Tagged PDF
minim-xmp	for XMP (metadata) inclusion

The documentation for the above packages will be replicated in separate chapters below.

You can use this package by simply saying `\input minim`; this will load the file minim.tex. For building your own format files, you can re-use the file minim.ini: if you define `\fmtname` before inputting this file, no `\dump` will be performed.

Contents

Compatibility

One central design goal of minim is to be as unobtrusive as possible: you should be able to safely ignore any function you do not want to use. Please get in touch if you find this not the case. Particular care has been taken to be compatible with ltluatex. All overlapping functions should produce the same results and ltluatex can be loaded either before or after minim.

Licence

Metapost

This package offers low-level mplib integration for plain luatex. The use of multiple simultaneous metapost instances is supported, as well as running tex or lua code from within metapost. In order to use it, simply say `\input minim-mp.tex`.

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.

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).

The logs of the metapost run will be included in the regular log file. If an error occurs, the log of the last snippet will also be shown on the terminal.

As a stand-alone Metapost compiler

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.

LaTeX compatibility

An experimental latex package is included in `minim-mp.sty`. It really is a rather thin wrapper around the plain tex package, but does provide a proper `metapost` latex 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}`. If your demands are even more complex, you should fall back to the plain tex commands described in the next section.

As in luamplib, you can use `\mpcolor {name}` to insert the proper colour values; this macro is only available inside the above environments.

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 though, but it will define an `mplibcode` environment, as well as the `\mplibcodeinherit`, `\mplibshowlog`, `\mplibsetformat` and `\mplibnumbersystem` switches. Please do note that this is not the recommended way of using minim-mp, which remains the interface documented in the next section.

Metapost instances

For more complicated uses, you can define your own instances by saying `\newmetapostinstance [ options ] \id`. An instance can be closed with `\closemetapostinstance \id`. These are the options you can use:

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`	`0`	Catcode table for `btex ... etex`.
`env`	copy of `_G`	Lua environment; see below.

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.

`\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 regardless of its position, 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.

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

Running tex from within metapost

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.

Previously-defined box resources can be included with `boxresource(nr)`. 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.

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).

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`.

For the greatest amount of control, you can use the `glyph g of f` operator, which returns the contours that make up a glyph. 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 be a font id or font name (as with `infont`).

A variant of `glyph of` is the `contours t of f` macro: it first typesets the string `t` in the same way as `infont` does (so that kerning and font shaping are applied), but returns a (comma-separated) list of contours, such as may be used in a `for` loop. Due to rounding errors, the glyphs will not match exactly.

Be aware that the contours returned by these operators may be disjoint: a letter `o`, for example, will consists of two. This means you cannot recreate the characters by just filling each contour: this 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 the next section).

Partial paths and the even-odd rule

You can fill or draw two or more disjoint paths in one go by using `nofill` as drawing operator for all paths but the last. This may make it easier to cut something out of a shape, since you do not have to worry about stitching the paths together.

While metapost fills paths according to the winding number, the pdf format also supports filling according to the even-odd rule. This has been made possible with the `eofill` and `eofilldraw` drawing statements, which can of course also be used as the final statement after a series of nofills.

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.

Finally, two handy clipping macros have been added: `clipout` and `clipto`, which both receive a list of paths as a ‘text’ parameter and either clip their ensemble out of the picture, or the picture to the ensemble.

Running lua from within metapost

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. (You can of course order the global environment to be used by giving `env = _G` as option to `\newmetapostinstance`.) Any environment you specify will be supplemented with the contents of the `M.mp_functions` table. It currently contains two functions: `quote(s)`, which escapes all double quotes in the string `s` before surrounding it with the same (so that it may be read as a metapost string literal); and `sp_to_pt(nr)`, which prints dimensions in points (preventing overflows).

When using `runscript` in this way, 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.

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 metapost. Numbers and strings will be returned as they are (so make sure the string is surrounded by quotes if you want to return a metapost string). You can return a point, colour or transform by returning an array of two to six elements (excluding five). For other return values, `tostring()` will be called.

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.

Tiling patterns

The condition `withpattern(<name>)` added to a `fill` statement will fill the path with a pattern instead of a solid colour. If the patterns contains no colour information of itself, 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.

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 `value`s 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 the scope of metapost.

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>` 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).

A transparency group is a special kind of XForm, and these can be created from withing metapost: `<id> = saveboxresource (<attributes>) <picture>` returns a number identifying the resource and can be fed attributes in the same way as `setgstate`. XForms defined through metapost are available to other metapost instances but not to tex, though the macro painting them, `boxresource <id>`, also accepts identifiers of tex-defined box resources. There remains a subtle difference, however: metapost-defined box resources are placed at their original origin.

Other 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.

Minim-mp also provides a few elementary macros and constants that are conspicuously absent from plain.mp; I hope their addition is uncontroversial. 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).

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, while `M.on_line` controls whether the logs are always printed to the terminal.

Debugging

You can enable (global) debugging by saying `debug_pdf` to metapost 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, a small summary of every generated image will be written to log and terminal.

Extending metapost

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.

The `M.init_code` table contains the code used for initialing new instances. In it, the string `INIT` will be replaced with the value of the `format` option (normally `plain.mp`).

Mathematics

This package gives a simple and higly-configurable way of using unicode and OpenType mathematics with plain LuaTeX, making use of most of the latter engine's new capabilities in mathematical typesetting. Also included are proper settings and definitions for nearly all unicode mathematical characters, as well as a few shorthands and helper macros that seemed useful additions.

Load the package by saying `\input minim-math.tex`; this will set up luatex for using opentype mathematical fonts and unicode math input. It will not, however, select mathematical fonts for you. That you will have to do for yourself; see below for instructions.

Styles and alphabets

For some (mostly alphabetical) characters, multiple variants are available, e.g. $A\mathrm{A}\mathbf{A}\boldsymbol{A}\mathbb{A}\mathfrak{A}\boldsymbol{\mathfrak{A}}\mathcal{A}\boldsymbol{\mathcal{A}}$. You can (locally) override the default style of these with `\mathstyle {style}` (equivalent to the old `\bf`, `\rm` etc.) or with one of the shorthands that apply the style to their argument only:

Shorthand	Value of `style`	Result
`\mup`	`up/rm`	ABC
`\mit`	`it`	ABC
`\mbf`	`bf`	$\mathbf{ABC}$
`\mbfit`	`bfit`	$\boldsymbol{ABC}$
`\mbb`	`bb`	$\mathbb{ABC}$
`\frak`	`frak`	$\mathfrak{ABC}$
`\bffrak`	`bffrak`	$\boldsymbol{\mathfrak{ABC}}$
`\scr`	`cal/scr`	$\mathcal{ABC}$
`\bfscr`	`bfscr`	$\boldsymbol{\mathcal{ABC}}$

Styles without shorthand are `sans/sf`, `sfit`, `sfbf`, `sfbfit`, `tt/mono` and finally the special value `clear` for using the default style. You can use the shorthands directly in sub- and superscripts: `v^\scr F` will result in $v^{\mathcal{F}}$.

While math families are not used anymore for switching between styles (see below), you still can use `\fam` with the values 0, 1, 2, 4, 5, 6 or 7 for doing so. This means that plain tex's `\rm`, `\it`, `\cal`, `\sl`, `\bf` and `\tt` can still be used (at least in math mode).

Please note that `\mup` is not the right choice for upright multiletter symbols or operators: you should use `\mord` or `\mop` instead (see near the end of this chapter). For nonmathematical text, you should use `\text` instead of `\mup`.

The default properties of characters can be set with one of the following three commands:

```
\mathmap   {character list} {style}
\mathclass {character list} {class}
\mathfam   {character list}  nr
```

There, `style` is one of the above and `class` is the name of a class as below. Finally, the `character list` should be a comma-separated list with elements of one of the following forms:

1. a list of characters, like `abc` or `\partial` or ℝ;
2. a character range, like `` `A-`Z ``, `65-90` or `"41-"5A`;

3. one of the alphabets `[Ll]atin`, `[Gg]reek`, or `digits`;
4. one of the style groups `bold`, `boldgreek`, `sans`, `sansgreek`, `mono`, `blackboard`, `fraktur` or `script`;
5. the name of a math class: `ord`, `op`, `bin`, `rel`, `fence`, `open`, `close` or `punct`.

Note that unicode is somewhat irregular in its encoding of mathematical letters; this is taken into account when using ranges as under (2) above. Thus, ``\mscra-`\mscrz`` really gives you all lowercase script characters, despite e.g. e being well outside that range.

The default style settings are `\mathmap {latin, greek, Latin}{it}`. Since the math family setting is not used anymore for selecting different styles, the default family of every symbol is zero. Instead, you can use `\mathfam` for mixing fonts (see below). The `class` option to `\mathclass` should be one of the names under 5.

Character variants

You can change the default appearance of several greek characters with `\usemathvariant {chars}` or `\usemathdefault {chars}`, where `chars` is a list of normal greek characters. As in unicode but against tex's tradition, the variants are $\epsilon\vartheta\Theta\varkappa\varpi\varrho\phi$ and the defaults $\varepsilon\theta\Theta\kappa\pi\rho\varphi$. The macros `\varepsilon` etc. have been updated to reflect the unicode variants.

The appearance of root symbols can be set with `\closedroots` ($\sqrt{2}$) and `\normalroots` ($\sqrt{2}$, the default).

Say `\unicodedots` to use the unicode dots characters (…⋮⋰⋱) and `\traditionaldots` to construct these characters from periods ($\ldots\vdots\cdot^{\cdot^{\cdot}}\,{}^{\cdot}{}_{\cdot}{}_{\cdot}$, the default). Both settings affect the meaning of both the actual characters and the `\xdots` macros ($\mathtt{x} \in \{\mathtt{l}, \mathtt{v}, \mathtt{c}, \mathtt{a}, \mathtt{d}\}$). Unlike in traditional plain tex, the traditional dots are available in script sizes, too.

Say `\decimalcomma` and have commas appear as 1,2 instead of $1, 2$ (`\nodecimalcomma` restores the default). Say `\smartdecimalcomma` for a comma that only acts as punctuation when not immediately followed by a digit. The explicit `\comma`, like `\colon`, will always be punctuation.

The behaviour of limits on integral signs can be set by redefining `\intlimits` (the default is `\let \intlimits = \nolimits`).

If you want to change the meaning (inside math mode) altogether for a single character, you can use the commands `\mathdef` and `\mathlet`. For example, by default, you can use the letter ħ for the reduced planck constant $\hbar$; this has been made so with `\mathdef ħ {\hbar}` (you could also have said `\mathlet ħ \hbar`).

Setting up fonts

The minimum you need do to set up a mathematical font is this:

```
\font\tenmath
    {Latin Modern Math:mode=base;script=math;ssty=0;} at 10pt
\font\tenmaths
    {Latin Modern Math:mode=base;script=math;ssty=1;} at 7pt
\font\tenmathss
    {Latin Modern Math:mode=base;script=math;ssty=2;} at 5pt
```

```
\textfont         0 = \tenmath
\scriptfont       0 = \tenmaths
\scriptscriptfont 0 = \tenmathss
```

Note that you only have to set up the font for a single family: opentype mathematical fonts typically contain all necessary variants of all mathematical characters. Therefore, the `\fam` setting has been made a no-op (use `\setfam` if you really need the old primitive) and the default family of all symbols has been set to zero.

As mentioned above, you can still change the family number of specific characters and this allows you to mix mathematical fonts. For instance, if you dislike the current blackboard bold characters, just assign a second font to family 1 and say `\mathfam {blackboard} 1`. Less useful are the parameters `\accentfam`, `\radicalfam` and `\extensiblefam` that control the family of all accents, radicals and extensibles.

Do note that various spacing constants are set according to the *last* math family that is assigned to. Therefore, you should assign your main math font to a family after all others.

Shorthands and additions

You can use `\text` for adding nonmathematical text to your equations. It will behave well in sub- and superscripts: `\text{word}^\text{word}` gives $\text{word}^{\text{word}}$. By default, the font used is the normal mathematical font. You can change this by setting the `\textfam` parameter to some nonzero value and assigning a different font to that family (see above). You probably want to do this, since most commonly-used mathematical fonts do not include a normal kerning table.

All the usual arrows can be made extensible by prefacing them with an `x`, including `\xmapsto` and `\xmapsfrom`. Alternatively, you can use `\→` etc. as shorthands. Additionally, you can use the following:

Shorthand	Result
`\bra x, \ket y`	$\langle x\rvert, \lvert y\rangle$
`\braket x y`	$\langle x\vert y\rangle$
`\norm x, \Norm x`	$\lvert x\rvert, \lVert x\rVert$
`\floor x, \ceil x`	$\lfloor x\rfloor, \lceil x\rceil$
`x \stackrel ?= y`	$x \stackrel{?}{=} y$
`x \stackbin a+ y`	$x \overset{a}{+} y$
`f\inv`	f^{-1} (cf. f^{-1})
`a \xrightarrow[down]{up} b`	$a \xrightarrow[down]{up} b$
`a \xeq[down]{up} b`	$a \overset{up}{\underset{down}{=\!=}} b$
`\frac12, \tfrac12, \dfrac12`	$\frac12, \tfrac12, \dfrac12$

Also new are the operators `\Tr`, `\tr`, `\Span`, `\GL`, `\SL`, `\SU`, `\U`, `\SO`, `\O`, `\Sp`, `\im`, `\End`, `\Aut`, `\Dom` and `\Codom`. You can define new operators with `\newmathop` and `\newlargemathop`: `\newmathop{op}` will define the new operator `\op`. For single use of an upright symbol, operator or large operator you can use `\mord`,

`\mop` and `\mlop`. The difference between `\mord` and `\mup` is that `\mord` also applies the correct symbol spacing.

The accents `\overbrace`, `\underbracket` etc. allow a label between square brackets: `$$\underbrace[=1]{(x^2+y^2)}$$` gives

$$\underbrace{(x^2+y^2)}_{=1}.$$

Finally, the following (entirely optional) alternative to using dollar signs is provided, which also offers slight improvements in the spacing of displayed equations:

	`\[ ... \]`	inline mathematics
`\display`	`\[ ... \]`	display mathematics
`\displaynr {nr}`	`\[ ... \]`	numbered display mathematics
`\displaynr`	`\[ ... \]`	automatically numbered display mathematics

The automatic display numbering uses the count `\equationnumber` and the token list `\setequationnumber` internally. All displays created this way can be made left-aligned by saying `\leftdisplaystrue`.

Best practices

The following remarks on mathematical typesetting have no relation to the contents of this package; I have included them because I find them hard to remember.

1. `\eqalign` gives a vertically centered box and can occur many times in an equation, while `\eqalignno` and `\leqalignno` span whole lines (put the equation numbers in the third column). All assume the relation (or operator) appears at the right hand side of the ampersand.
2. The command `\displaylines` can only have one column that spans the whole line (and you will have to add the equation number by hand).
3. Further alignment commands are `\cases`, `\matrix`, `\pmatrix` (with parentheses) and `\bordermatrix` (includes labels for lines and columns).
4. Finetuning alignments can be done with `\smash`, `\phantom`, `\hphantom` and `\vphantom`.
5. Small matrices like $\left({1\ 2 \atop 3\ 4}\right)$ can be made by misusing `\choose` or `\atop`.
6. If you start a line with a binary operator, put a `{}` before it: this way, tex recognises it as such.
7. Thin spaces (`\,`) should be inserted: before dx, before units, after factorials and after `\dots` if those are followed by a closing parenthesis.
8. Whether the differential operator should be set upright or not is as of yet an open question in mathematics.
9. You should prefer `\bigr` and `\bigl` etc. over `\big`, `\Big`, `\bigg` and `\Bigg`.
10. An overview of mathematical symbols, with control sequences and their availability in different fonts, can be found in `unimath-symbols.pdf`, which is part of the unicode-math package.

Advanced PDF features

This chapter and the next document the support of the modern pdf features provided by the minim-pdf package. Load it by saying `\input minim-pdf`. The next chapter concerns the creation of tagged pdf; all other features of the package are described here.

Hyperlinks

Hyperlinks can be made with `\hyperlink [alt {...}] [attr {...}] <action> ... \endlink`, where the `<action>` must be one of `name {...} | url {...} | name {...} | next | prev | first | last` With the `name` action, a named destination must be used (see below), while the `user` action will be passed directly to the back-end (as with the pdftex primitive). After the `url` action, the characters ~, # and % need not be escaped. (Of course, this does nothing for already-tokenised text; be aware of this when you wrap `\hyperlink` into another macro.) Any spaces after the `<action>` will be ignored.

The `\hyperlinkstyle` token list can be used so set common (pdf) link attributes; it defaults to `/Border [0 0 0]`. The contents of the optional `attr` parameter will be appended to these. The `alt` options sets the `/Contents` key that is required by PDF/UA (where it has the purpose of an alt text).

A named destination can be created with `\nameddestination {...}` (also in horizontal mode, unlike the backend primitive) and if you cannot think of a name, `\newdestinationname` should generate a unique one. If you need the latter twice, `\lastdestinationname` gives the last generated name.

Bookmarks

Bookmarks (also known as outlines) can be added with `\outline [level n] [open|closed] [dest {name}] {title}`. Add `open` or `closed` to have the bookmark appear initially open or closed (the default), and say `dest {name}` for having it refer to a specific named destination (otherwise, a new one will be created where the `\outline` command appears).

In the absence of the `level` option, the bookmark is automatically associated with the current structure element and the hierarchy of structure elements determines the nesting of bookmarks. This works even if you have otherwise disabled tagging and is the recommended way of generating outlines. (You can find all relevant macros in next chapter under 'Document structure' and 'Structure element aliases'.)

As a fallback, outlines specified with the `level n` option will be inserted at the end of the current outline list at the specified level ($n \geq 1$ and need not be contiguous). Both methods can be intertwined, but please use the document structure if you can.

Page labels

If the page numbers of your document are not a simple sequence starting with 1, you can use `\setpagelabels [pre {prefix}] style nr` for communicating this to the pdf viewer. This command affects the page labels from the next page on: `nr` should be the numerical page number of that page. The `prefix` is prepended to each number and the `style` must be one of `decimal`, `roman`,

Roman, alphabetic, Alphabetic or none. In the last case, only the prefix is used.

PDF/A

You can declare pdf/a conformance with `\pdfalevel xy`, with version $x \in \{1, 2, 3\}$ and conformance level $y \in \{a, b, u\}$. This will set the correct pdf version and pdfaid metadata. If the conformance level is ‘a’, tagging will be enabled (see the next chapter). Finally, a default RGB colour profile will be included. The conformance level can be queried from the `\pdfaconformancelevel` register.

Note that merely declaring conformance will not make your document pdf/a compliant, and that minim will not warn you if it is not. However, the features described in this chapter and the next should be enough to make pdf/a compliance possible.

Also note that there currently is no documented way of choosing a different colour profile from the default (i.e. the default rgb profile provided by the colorprofiles package). Should you need do that, you will have to do so manually, after disabling the automatic inclusion by saying `\expandafter\let \csname minim:default:rgb:profile\endcsname = \relax`.

Finally, note that pdf/a requires that spaces are represented by actual space characters and that discretionary hyphens are marked as soft hyphens (U+00AD). Since both features benefit accessibility and text extraction in general, they are enabled by default. You can disable them by setting `\writehyphensandspaces` to a nonpositive value.

PDF/UA

You can claim pdf/ua conformance with `\pdfualevel 1`. By itself, this will do very little:

1. The pdfuaid:part metadata key will be set.
2. A conforming ViewerPreferences dictionary will be added to the document catalog.
3. The /Suspects key of the MarkInfo dictionary will be set to false.
4. /Tab /S will be added to the page attributes.

Also making your document pdf/a-compliant, however, will relieve you of a few additional worries:

5. Fonts will be included properly.
6. The (natural) language of every element will be known.
7. Headings will be strongly-structured.
8. Table headers will have their Scope set properly.
9. A document outline will be generated automatically.

This leaves the following for you to provide before your document can be pdf/ua-compliant:

10. Figure and Formula structure elements must have alt texts.
11. Hyperlinks must have alternate descriptions.
12. Lists must have the ListNumbering attribute set.
13. Tables must have headers that are tagged as such.

14. Page headers and footers must be marked as header or footer artifacts.
15. Document section structure elements should have their `title` set.
16. All embedded files must have a description.

Embedded files

You can attach (associate) files with `\embedfile <options>`. The file will be attached to the current structure element (see the next chapter) unless the `global` option is given: then it will be added to the document catalog. Arguments consisting of a single word can be given without braces and exactly one of the options `file` or `string` must be present.

`file`	`{...}`	The file to embed.
`string`	`{...}`	The string to embed.
`global`		Attach to the document catalog.
`uncompressed`		Do not compress the file stream.
`mimetype`	`{...}`	The file's mime type.
`moddate`	`{...}`*	The modification date (see * below).
`desc`	`{...}`	A description (the `/Desc` key).
`relation`	`{...}`	The `/AFRelationship` value as defined in pdf/a-3.
`name`	`{...}`	The file name (only required when writing a `string`).

* The modification date must be of the form `yyyy[-m[m][-d[d]]]`. A default moddate can be set with `\setembeddedfilesmoddate {default}`. The `default` date will be expanded fully at the time of embedding. With the minim-xmp package, a useful setting is `\setembeddedfilesmoddate {\getmetadata date}`.

Lua module

The interface of the lua module (available via `local M = require('minim-pdf')`) should be stable by now. Though it contains lua equivalents for most tex commands described here, using them directly is not very ergonomical and not recommended. Please consult the source if you do want to use them anyway.

Tagged PDF

This chapter is a continuation of the previous and describes the parts of minim-pdf that concern the creation of tagged pdf. All features in this chapter must be explicitly enabled by setting `\writedocumentstructure` to a positive value. This will be done automatically if you declare pdf/a conformance (see above).

This part of the package is rather low-level and this chapter rather technical. For a more general introduction to and discussion of tagged pdf, please read the (excellent) manual of latex's tagpdf package.

Quick-start guide

The minimal setup for producing tagged pdf from plain tex documents is something like the following:

```
% first update all fonts to TrueType (ttf) or OpenType (otf)
\input luaotfload.sty
% ... font redefinitions omitted ...
\input minim-mp
\pdfalevel 2a % declare pdf/a conformance, enable tagging
\autotagplainoutput % update the output routine
% create section markers and counters
\sectionstructure { subsection, section, chapter }
```

You can then update your sectioning macros to look like this:

```
\def\section#1\par{%
    % space above
    \bigskip \goodbreak
    % structure and outline (this is the new part)
    \marksection \outline open {#1}
    % section header
    \noindent {\bf \the\chapternr.\the\sectionnr. #1}
    % space below
    \smallskip \noindent}
```

Other macros you might have for laying out structural elements, such as tables or lists, should of course also be updated. The rest of this chapter describes the tools you can use.

Please be advised that producing tagged pdf will likely forever remain a fragile and error-prone process. You should always validate the resulting pdf. An easy-to-use and free validator is veraPDF. For inspecting the document structure, you can use the `pdfinfo` utility that comes with the Poppler pdf library.

Purpose, limitations and pitfalls

The main purpose of this package is semi-automatically marking up the (hierarchical) structure of your document, thereby creating so-called tagged pdf. The mechanism presented here is not quite as versatile as the pdf format allows. The most important restriction is that all content of the document must be seen by tex's stomach in the *logical* order.

Furthermore, while the macros in this package are sophisticated enough that tagging can be done without any manual intervantion, it is quite possible and rather easy to generate the wrong document structure, or even cause syntax errors in the resulting pdf code. You should always inspect and validate the result.

This is the full list of limitations, pitfalls and shortcomings:

1. Document content must be seen by tex in its logical order (although you can mark out-of-order content explicitly if you know what you are doing; see below).
2. The contents of `\localleftbox` and `\localrightbox` must be marked manually, probably as artifact.
3. There currently is no way of marking xforms or other pdf objects as content items of themselves.
4. The content of xforms (i.e. pdf objects created by `\useboxresource`) should not contain tagging commands.
5. Likewise, you should be careful with box reuse: it might work, but you should check.
6. This package currently only supports pdf 1.7 tagging and is not yet ready for use with pdf 2.0.

In order to help you debugging, some errors will refer you to the resulting pdf file. If you get such errors, decompress the pdf and search for the string 'Warning:'. It will appear in the pdf stream at the exact spot the problem occurs.

General overview

When speaking about tagging, we have to do with two (or perhaps three) separate and orthogonal tagging processes. The first is the creation of a hierarchical *document structure*, made up of *structure elements* (SEs). The document structure describes the logical structure of a document, made up of chapters, paragraphs, references etc. The second tagging process is the tagging of *marked content items* (MCIs): this is the partition of the actual page contents into (disjoint) blocks that can be assigned to the proper structure element. Finally, as a separate process, some parts of the page can be marked as *artifacts*, excluding their content from both content and structure tagging.

When using this package, artifacts and structure elements (excluding paragraphs; see below) must be marked explicitly, while marked content items will be created, marked and assigned automatically. There is some (partial and optional) logic for automatically arranging structure elements in their correct hierarchical relation.

The mechanism through which this is achieved uses attributes and whatsits for marking the contents and borders of SEs, MCIs and artifacts. At the end of the output routine, just before the pdf page is assembled, this information will be converted into markers inserted in the pdf stream.

Marked content items

Content items are automatically delineated at page, artifact and structure element boundaries and terminated at paragraph or display skips. This should relieve you from any manual intervention. However, if you run into problems, the commands below might be helpful.

Use of `ActualText`, `Alt` or `Lang` attribute on MCIs, while allowed by the pdf standard, is not supported by this package. You should set these on the structure element instead.

The beginning and ending of a content item can be forced with `\start-contentitem` and `\stopcontentitem`, while `\ensurecontentitem` will only open a new content item if you are currently outside any. If you need some part to be a single content item, you can use `\startsinglecontentitem ... \stopsinglecontentitem`. This will disable all SE and MCI tagging inside.

Tagging (both of MCIs and SEs) can be disabled and re-enabled locally with `\stoptagging` and `\starttagging`.

Artifacts

Artifacts can be marked in two ways: with `\markartifact {type} {...}` or with `\startartifact {type} ... \stopartifact`. The `type` is written to the pdf attribute dictionary directly, so that if you need a subtype, you can write e.g. `\startartifact {Pagination /Subtype/Header} etc`.

Inside artifacts, other structure content markers will be ignored. Furthermore, this package makes sure artifacts are never part of marked content items, automatically closing and re-opening content items before and after the artifact. While the pdf standard does not require the latter, not enforcing this seems to confuse some pdf software.

Document structure

Like artifacts, structure elements can be given as `\markelement {Tag} {...}` or `\startelement {Tag} ... \stopelement {Tag}`. Here, in many cases the `\stopelement` is optional: whenever opening an element would cause a nesting of incompatible `Tag`s, the current element will be closed until such a nesting is possible. Thus, opening a `TR` will close the previous `TR`, opening an `H1` will automatically close any open inline or block structure elements, opening a `TOCI` will close all elements up until the current `TOC` etc. etc.

As a special case, the tags `Document`, `Part`, `Art`, `Sect` and `Div` (and their aliases) will try and close all open structure elements up to and including the last structure element with the same tag. (An alias will of course only match the same alias.)

While the above can greatly reduce the effort of tagging, the logic is neither perfect nor complete. You should always check the results in an external application. Particular care should be taken when 'skipping' structure levels: the sequence chapter – subsection – section will result in the section beneath the subsection. If you are in doubt whether an element has been closed already, you can use `\ensurestopelement {Tag}` instead of `\stopelement` to prevent an error being raised.

All these helpful features can be disabled by setting `\strictstructuretagging` to a positive value. Then, every structure element will have to be closed by an explicit closing tag, as in xml. In this case, `\stopelement` and `\ensurestopele-ment` will be equivalent.

You can query the place in the document structure of any point with `\show-documentstructure`.

Structure element aliases

New structure element tags can be created with `\addstructuretype [options] Existing Alias`. This will create a new structure tag named `Alias` with the same properties as `Existing`. The properties can be modified by specifying `options`: these will set values of the corresponding entry in the `structure_types` table (see the lua source file for this package). Any aliases you declare will be written to the pdf's `RoleMap` only if they have actually been used.

Automatic tagging of paragraphs

By default, `P` structure elements are inserted automatically at the start of every paragraph. The tag can be changed with `\nextpartag {Tag}`; leaving the argument empty will prevent marking the next paragraph. Keep in mind that the (internal) reassignment is local: if a paragraph marked with `\nextpartag` starts inside a group, it will not reset. Hence, to avoid surprises, you should have `\nextpartag` and the start of your paragraph at the same grouping level.

Useful structure elements for `\nextpartag` include `H` for headings and `LI` for list items. Since minim-pdf produces strongly-structured documents, the tags `H1`, `H2`, `H3` etc. should not be used.

Please also note that if you add `\hbox`es directly to a vertical list (this includes `\line`, `\centerline` and the like), the `\everypar` token list is not inserted and no new paragraph structure element will be opened. The contents of the `\hbox` will be added to the current structure element, and this may result in an invalid structure hierarchy (and an error messsage reading 'Structure type mismatch'). You can make your intentions clear by inserting `\startelement{P}` at the appropriate place (see above).

Auto-marking paragraphs can be (locally) disabled or enabled by saying `\markparagraphsfalse` or `\markparagraphstrue`.

Manipulating the logical order

With the process outlined above, the logical order of structure elements has to coincide with the order in which the SEs are 'digested' by tex. This, together with the marked content items being assigned to structure elements in their order of appearance, lies behind the restriction that logical and processing orders should match.

With manual intervention, this restriction can be relaxed somewhat. Issuing the pair `\savecurrentelement ... \continueelement` will append the MCIs following `\continueelement` to the SE containing `\savecurrentelement`. Since the assignments made here are global, this process cannot be nested; in more complicated situations you should therefore use `\savecurrentelementto\name ... \continueelementfrom\name` which restores the current SE from a named identifier `\name`.

Structure element options

The `\startelement` command allows a few options that are not mentioned above: its full syntax is `\startelement <options> {Tag}`. The three most useful options are `alt` for setting an alt-text (the `/Alt` entry in the structure element dicionary), `actual` for a text replacement (`/ActualText`) and `lang` for the language (`/Lang`; see the next section). The alternative and actual texts can also be given after the fact with `\setalttext {...}` and `\setactualtext {...}`; these apply to the current structure element.

Structure element attributes can be given with `attr <owner> <key> <value>`, e.g. `attr Layout Placement /Inline` or added later with `\tagattribute`. Note that for the `owner` and `key` the initial slash must be omitted; the `value` on the other hand will be written to the pdf verbatim. Any number of attributes can be added.

An identifier can be set with the `id {...}` option, or after the fact with `\settagid {...}`. This identifier will be added to the `IDTree` and is entirely optional; you will probably already know when you need it. The `ref {...}` option lets a structure element refer to another (the `/Ref` option in the structure element dictionary). Its argument should be the `id` of the other structure element.

The title of the structure element (corresponding to the `/T` entry in the structure element dictionary) can be set with the `title {...}` option. The pdf/ua standard requires this key for all document sections.

Finally, structure element classes can be given with the `class <classname>` keyword, which can be repeated. Classes can be defined with `\newattributeclass classname <attributes>` where `<attributes>` can be any number of `attr` statements as above.

Languages

If you do not specify a language code for a structure element, its language will be determined automatically. In order for this to work, you must associate a language code to every used language; you can do so with `\setlanguagecode name code`, where `name` must be an identifier used with `\uselanguage {name}` and `code` must be a two or three-letter language code, optionally followed by a dialect specification, a country code, and/or some other tag. Note that the language code is associated to a language *name*, not to the numerical value of the `\language` parameter. This allows you to assign separate codes to dialects.

There is a small set of default language code associations, which can be found in the file `minim-languagecodes.lua`. It covers most languages defined by the hyph-utf8 package, as well as (due to their ubiquitous use) some ancient languages.

An actual language change introduced by `\uselanguage` will not otherwise be acted upon by this package. Therefore, you will probably want to add `\startelement{Span}` after every in-line invocation of `\uselanguage`.

You can set the document language with `\setdocumentlanguage languagecode`. If unset, the language code associated with the first `\uselanguage` statement will be used, or else `und` (undetermined). The only function of the document language is that it is mentioned in the pdf catalog: it has no other influence.

New languages can be declared with `\newnamedlanguage {name} {lhm} {rhm}` and new dialects with `\newnameddialect {language name} {dialect name}`. Dialects will use the same hyphenation patterns (and will indeed have the same `\language` value) as their parent languages; newly declared languages will start with no hyphenation patterns. Do note that you will probably also have to specify language codes for new languages or dialects.

This package ensures the existence of the `nohyph`, `nolang`, `uncoded` and `undetermined` dummy languages, all without hyphenation.

Mathematics

You can auto-tag equations as formulas by specifing `\autotagformulas`. After this command, auto-tagging can be switched off and on with `\stopformulatagging` and `\startformulatagging`. Auto-tagging formulas is dangerous, because sometimes equations are used for lay-out and should not be marked as such. It is also somewhat fragile, as it requires equations to end with dollar signs (and not with `\Ustopmath` or `\Ustopdisplaymath`).

The tex source of an equation can be associated with the `Formula` structure element in various ways, which can be configured with `\includeformulasources {options}`, where the `options` must be a comma-separated list of `alttext`, `actualtext` or `attachment`. The `alttext` and `actualtext` option will set the `/Alt` or `/ActualText` attributes to the unexpanded source code of the equation, surrounded by the appropriate number of dollar signs. The `attachment` option attaches the source of the formula as an embedded file with its `/AFRelation` set to `Source`; this will only work if `\pdfaconformancelevel` equals three. The name of this file can be changed by redefining `\formulafilename` inside the equation. The default value is `{actualtext,attachment}`.

Note that the contents of the equation will be expanded fully (as in `\xdef`) before their inclusion as the equation source. This may place restrictions on the macros you want to use (those in minim-math should be safe). Any occurrence of `\alttext` or `\actualtext` overrides the automatically-assigned value and will be stripped from the equation source.

Tables

For marking up tables, a whole array of helper macros is available. First, `\marktable` should be given *before* the `\halign`. Then, in the template, the first cell should start with `\marktablerow \marktablecell` and each subsequent cell with `\marktablecell`. If your table starts with a header, insert `\marktableheader` before it and `\marktablebody` after. Before a table footer, insert `\marktablefooter`.

For greater convenience, insert just `\automarktable` before the `\halign`. Then you can leave out all the above commands (unless you `\omit` a template of course). This assumes the table has a single header row and more than one column. If you use a table for typesetting a list, you can use `\marktableaslist` instead, which marks the first column as list label and the second column as list item. Of course, this only works with two-column tables.

Cells spanning multiple cells or rows can be marked with `\markcolumnspan {width}` and `\markrowspan {height}`; these statements may not occur before `\marktablecell`. Note that while `\markcolumnspan` properly increases the (internal) column number, `\markrowspan` does nothing of the sort (and indeed, no general logic can be given in the latter case). Always proceed with caution when using cells spanning multiple rows, and inspect the resulting structure carefully.

Marking up a table header (except if done through `\automarktable`) will not connect normal table cells with their headers; you will have to connect these manually by including `\markcolumnhead` or `\markrowhead` in the appropriate header cells. This must be done *after* `\markcolumnspan` if the latter appplies.

If properly setup like this, other cells of the table (including header cells) will be assigned to matching row or column headers automatically.

Other helper macros

For marking up an entry in a table of contents, you can use the macro `\marktocentry {dest} {lbl} {title} {filler} {pageno}`, which should insert all tags in the correct way. (The `dest` is a link destination and can be empty; the `lbl` is a section number and can also be empty.)

For tagging (foot)notes, `\marknoteref{*}` and `\marknotelbl{*}`, when placed around the footnote markers, will insert the proper `Ref`, `Note` and `Lbl` tags.

Helper macros for tagging sections can be setup quickly with `\sectionstructure { <section list> }`. The `<section list>` should be an increasing comma-separated list of section types, e.g. `{subsection, section, chapter}`. This will first reserve the `\count` registers `\subsectionnr` etc, then create the structure aliases `/Subsection` etc. and finally define the helper macros `\marksubsection` etc, which will do the following:

1. Call `\ensurestopelement` on all lower section types.
2. Set all lower section number counts to zero.
3. Increase the current section type number by one.
4. Call `\startelement` for the current section type.
5. Set the `\nextpartag` to `H`.

The proper place for these helper macros is just before the section heading; inbetween those two may come an `\outline` statement (see the previous chapter). You can set the `title` option to the internal `\startelement` statement with an optional argument (e.g. `\marksection [Section \the\sectionnr]`).

Tagging the output routine

The command `\autotagplainoutput` will try and update plain tex's output routine to produce tagged page artifacts and footnotes. It redefines `\makeheadline`, `\makefootline`, `\footnoterule`, `\footnote` and `\vfootnote`. Headline and footline will not be marked as artifacts if their contents equal `\hfil`; the footnote macros are edited to include the `\marknoteref` and `\marknotelbl` macros described above. Note that the `\topinsert`, `\midinsert` and `\pageinsert` macros are left untouched; you will have to mark those explicitly.

You can make some changes to the affected macros before calling `\autotagplainoutput`, as it tries to be smart about it. Though the redefinitions involve a full expansion, most conditionals and common typesetting instructions (`\line`, `\quad`, `\strut` etc.) are safe-to-use and will not be expanded. If you include custom macros of your own, however, it is wise to have those `\protected`.

Metadata

This package enables simple XMP (eXtensible Metadata Platform) packet inclusion and will automatically generate pdf/a extension schemas. Use it by saying `\input minim-xmp.tex`. Thereafter, you can use `\setmetadata key {value}` and `\getmetadata key` for setting and retrieving document-level metadata values.

You do not need this package if you have your metadata ready-made in a separate file, for then you can simply say

```
\immediate\pdfextension obj uncompressed
    stream attr {/Type/Metadata /Subtype/XML}
    file {your-file.xmp}
\pdfextension catalog
    {/Metadata \pdffeedback lastobj 0 R}
```

Setting metadata

Metadata fields that contain (ordered or unordered) lists will be split on the `\metadataseparator` character; this is a semicolon by default. Alternatively, you can just make multiple assignments: these will be concatenated.

Where applicable, language alternatives can be given like `\setmetadata /de dc:title {...}` or `\setmetadata /{de_DE} dc:title {...}`. Braces are necessary in the second case because the catcode of the underscore is not 11 or 12. When no alternative is given, the value `x-default` will be used.

Instead of using `\setmetadata`, multiple fields can be set in one go with `\startmetadata`. This way is particularly useful when assigning structured data to a key (see later on). In this example, `key1` contains a normal value, `key2` language alternatives and `key3` structured data:

```
\startmetadata
    key1 {...}
    key2 {... (default) ...}
        /alt1 {...}
        /alt2 {...}
    key3
        /field1 {...}
        /field2 {...}
    stopmetadata
```

Since metadata values are read by lua as text, linebreaks and paragraphs are not preserved. You can work around this by saying `{\def\par{\Uchar"A\Uchar"A} \setmetadata abstract {...}}`.

Getting metadata

Metadata values can be retrieved again with `\getmetadata key`. This command is fully expandible.

If the data is a list, it will be returned according to the current value of `\metadataseparator`. If the data has language alternatives, the `x-default` value will be returned: the alternatives are accessible by `\getmetadata /lang key`.

For structured types (discussed below), `\getmetadata /field key` will return the value of a single field and `\getmetadata key` will return all fields as `/{field1} {value1} /{field2} {value2} ...` (this can be used again as input to `\startmetadata`).

Supported metadata keys

Initially, the `\setmetadata` and `\getmetadata` recognise all pdf/a compatible fields in the `pdf`, `pdfaid`, `pdfuaid`, `dc`, `xmp`, `xmpMM` and `xmpRights` namespaces. Keys should be prefixed with their namespace, e.g. `dc:creator` or `xmp:CreatorTool`. Note that the `dc` namespace has lower-case fields; field names are case-sensitive.

Because the precise details of the above metadata namespaces can be confusing, you might want use one of the aliases `author` (dc:creator), `title` (dc:title), `date` (dc:date and xmp:CreateDate), `language` (dc:language), `keywords` (dc:subject and pdf:Keywords), `publisher` (dc:publisher), `abstract` (dc:description), `copyright` (dc:rights), `version` (xmpMM:VersionID) and `identifier` (dc:identifier). New aliases can be defined in the `aliases` table of the lua module.

Adding new keys and schemas

New metadata namespaces can be added in the following way:

```
require('minim-xmp').add_namespace(
  'Example namespace', 'colours',
  'http://example.com/minim/colours/', {
    -- metadata keys
    Favourite = { 'Colour', 'The author’s favourite colour' },
  }, {
    -- value types
    Colour = { 'RGB Colour', {
      R = { 'Integer', 'Red component' },
      G = { 'Integer', 'Green component' },
      B = { 'Integer', 'Blue component' }
    }, prefix = 'c' },
  })
```

This will setup a namespace with prefix `colours` and one key: `Favourite`, of type `Colour`. That value type happens to be a structured type with three fields, which are also described. You can now use this namespace as

```
\startmetadata colours:Favourite /R 0 /G 0 /B 255 stopmetadata
```

or the equivalent but more verbose

```
\setmetadata/R colours:Favourite 0
\setmetadata/G colours:Favourite 0
\setmetadata/B colours:Favourite 255
```

after which the generated XMP code will be

```
  <rdf:Description rdf:about=""
      xmlns:colours="http://example.com/minim/colours/"
      xmlns:c="http://example.com/minim/colours/">
    <colours:Favourite rdf:parseType="Resource">
      <c:B>255</c:B>
      <c:G>0</c:G>
      <c:R>0</c:R>
    </colours:Favourite>
  </rdf:Description>
```

If use of the pdf/a format is detected (i.e. a `pdfaid` entry is present in the metadata), the following pdf/a extension schema will also be generated:

```
<rdf:Description rdf:about=""
    xmlns:pdfaExtension="http://www.aiim.org/pdfa/ns/extension/"
    xmlns:pdfaSchema="http://www.aiim.org/pdfa/ns/schema#"
    xmlns:pdfaProperty="http://www.aiim.org/pdfa/ns/property#"
    xmlns:pdfaType="http://www.aiim.org/pdfa/ns/type#"
    xmlns:pdfaField="http://www.aiim.org/pdfa/ns/field#" >
  <pdfaExtension:schemas>
    <rdf:Bag>
      <rdf:li rdf:parseType="Resource">
        <pdfaSchema:schema>Example namespace</pdfaSchema:schema>
        <pdfaSchema:namespaceURI>http://example.com/minim/colours/</pdfaSchema:namespaceURI>
        <pdfaSchema:prefix>colours</pdfaSchema:prefix>
        <pdfaSchema:property>
          <rdf:Seq>
            <rdf:li rdf:parseType="Resource">
              <pdfaProperty:name>Favourite</pdfaProperty:name>
              <pdfaProperty:valueType>Colour</pdfaProperty:valueType>
              <pdfaProperty:category>external</pdfaProperty:category>
              <pdfaProperty:description>The author’s favourite colour</pdfaProperty:description>
            </rdf:li>
          </rdf:Seq>
        </pdfaSchema:property>
        <pdfaSchema:valueType>
          <rdf:Seq>
            <rdf:li rdf:parseType="Resource">
              <pdfaType:type>Colour</pdfaType:type>
              <pdfaType:namespaceURI>http://example.com/minim/colours/</pdfaType:namespaceURI>
              <pdfaType:prefix>c</pdfaType:prefix>
              <pdfaType:description>RGB Colour</pdfaType:description>
              <pdfaType:field>
                <rdf:Seq>
                  <rdf:li rdf:parseType="Resource">
                    <pdfaField:name>B</pdfaField:name>
                    <pdfaField:valueType>Integer</pdfaField:valueType>
                    <pdfaField:description>Blue component</pdfaField:description>
                  </rdf:li>
                  <rdf:li rdf:parseType="Resource">
                    <pdfaField:name>G</pdfaField:name>
                    <pdfaField:valueType>Integer</pdfaField:valueType>
                    <pdfaField:description>Green component</pdfaField:description>
                  </rdf:li>
                  <rdf:li rdf:parseType="Resource">
                    <pdfaField:name>R</pdfaField:name>
                    <pdfaField:valueType>Integer</pdfaField:valueType>
                    <pdfaField:description>Red component</pdfaField:description>
                  </rdf:li>
                </rdf:Seq>
              </pdfaType:field>
            </rdf:li>
          </rdf:Seq>
        </pdfaSchema:valueType>
      </rdf:li>
    </rdf:Bag>
  </pdfaExtension:schemas>
</rdf:Description>
```

You probably will not need defining your own value types, so in most cases you should omit the fifth argument to `add_namespace`. If you do define a new value type, you can specify its prefix if it differs from the namespace prefix (as is done above) and likewise its `uri` identifier if it differs from the namespace URI.

List types can be given as `'Bag TypeName'` or `'Seq TypeName'`; language alternatives as `'Lang Alt'`. All other types will be treated as `'Text'`, though for `'Boolean'`, `'Integer'` and `'Date'` some validation is performed when setting values.

Additional metadata value type handling can be defined in the `getters` and `setters` tables of the lua module. Appropriate entries to those tables will be generated automatically for new structured types (which is why you could set the colour like we did above). Value types without fields, however, will be stored and retrieved as if they were `Text` until you provide another way.

Generated XMP

All metadata will be written to the PDF file uncompressed.

The `\metadatamodification` setting controls whether XMP packets will be marked as read-only (value 0; default) or writeable (value 1). Writeable XMP packets will be padded with about 2 kB of whitespace. You can prohibit writing metadata altogether by specifying `\writedocumentmetadata = 0`.

If the document-level metadata contains values in the `pdfaid` namespace, metadata extension schemas will be appended to the document-level metadata packet automatically. These extension schema's will only include keys that have been set somewhere, though they need not have been set in the document-level metadata. No extension schemas are generated for the built-in namespaces, as they are already included in the pdf/a standards.

Programming macros

This chapter describes the programming helper modules on which all the above modules depend. It mainly concerns register allocation, callback management and format file inclusion.

They can be loaded separately by saying `\input minim-alloc`; thereafter, you can use `local M = require('minim-alloc')` to access the lua interface. In this chapter, when discussing lua functions, you are assumed to have issued the latter statement, so that the table `M` refers to the contents of this module.

The callback-related code lives in a separate file that can and must be loaded separately as `local C = require('minim-callbacks')`. This is the only file in this collection that does not itself depend on the minim-alloc module.

There is a large functional overlap between this module and the ltluatex package. You can use both at the same time, however, and the order in which you load both packages should not matter.

Format files

A major motivation for writing this module (and not, instead, depending on ltluatex) is the ability to write lua-heavy code that can be safely included in format files. For this purpose, the register allocation functions described below allow ensuring that the allocation is made only once.

Apart from registers, you need only do two more things to make your code format file safe. The first is saying `M.remember('your-file.lua')` somewhere, anywhere. This will mark your file for inclusion in the format. At the start of every job, all remembered files will be executed (in order) and their return values will be stored to be retrieved whenever you say `require('your-file')`. Note that while this feature does not improve speed in any meaningful way, it will ensure the lua file used by the format is identical to the one used to create it.

It does mean, however, that your file may be executed twice: once when building the format and once when the format is used. In most cases (e.g. callback registration) this is exactly what you want. Sometimes however, you may need to store variable (configurable) data in the format file. You can do this by saying `local t = M.saved_table('identifier', default-table)`. This will retrieve the table from the format file if possible; otherwise, it will return `default-table` and mark it to be saved to the format. A missing second argument is equivalent to an empty table. Saved tables may only contain (arbitrary but non-cyclic nestings of) tables, numbers and strings.

Register allocation

For allocating the new luatex registers, you can use the following: `\newfunction`, `\newattribute`, `\newwhatsit`, `\newluabytecode`, `\newluachunkname`, `\newcatcodetable` and `\newuserrule`. Note the one difference with ltluatex, which has `\newluafunction` instead. (The reason for this is that ltluatex, instead of a more sensible method, uses this macro for determining whether it has been read before.) Internally, the very same counts are used for keeping track of register allocation as in ltluatex. Their effect should therefore be identical in all circumstances, with one exception: no bounds checking is performed on any allocation macro defined by minim. Please do not go and use more than sixty five thousand different whatsits.

All the above and all traditional registers can be allocated from within lua as well, using `M.new_count('name')`, `M.new_whatsit('name')` etc. All return the allocated number. The (optional) string `name` prevents the same allocation from being made twice: if another register has been retrieved with the same name, the number of that register will be returned. You will need this when you want to allow your lua code to be included in a format file; it has nothing to do with the tex-side `\countdef` and the like.

If you want to access from lua a register defined in tex, the function `M.registernumber('name')` will give you the index of register `\name`.

Besides `\newluachunkname\name`, you can also use `\setluachunkname \name {actual name}` to enter the value of the name directly.

Finally, for the registers for which etex defines a local allocation macro (and for those only), you can use `M.local_count()` etc. These allocation functions take no parameters.

Callbacks

As noted at the beginning of this chapter, the callback functions are only available after you say `local C = require('minim-callbacks')`.

The simple function of this module is allowing multiple callbacks to co-exist. Different callbacks call for different implementations, and some callbacks can only contain a single function. Its interface matches the primitive interface, with `C.register(callback, fn)`, `C.find(callback)` and `C.list()` taking the same arguments. In addition to these, `C.deregister(fn)` will allow you to remove a callback. This is necessary when you want to remove a callback from a list or from the bottom of a stack. The `fn` variable should point to the same object.

Any callbacks that are already assigned before loading this module will be preserved and the primitive callback interface is still available, though callbacks registered through the latter will actually use the new functions. Ltluatex may be loaded either before or after this module.

You can create your own callbacks with `C.new_callback(name, type)`. The `type` should be one of the types mentioned below or `'single'` for callbacks that allow only one function. If the `name` is that of a primitive callback, new registrations will target your new callback. You can call the new callback with `C.call_callback(name, ...)`, adding any number of parameters.

Callbacks of type `node` operate on a node list: for these, all registered functions will be called in order, each function receiving the result of the last. After one function returns `false`, no others will be called. Callbacks of this type are `pre_linebreak_filter`, `post_linebreak_filter`, `hpack_filter`, `vpack_filter`, `pre_output_filter` and `mlist_to_mlist`.

Similarly, for the `data` callbacks `process_input_buffer`, `process_output_buffer` and `process_jobname`, all registered functions will be called in order on the output of the previous. Returning `false` will in this case result in the output of the previous function passing to the next.

For `stack` callbacks, a stack is kept and only the top function on the stack will be called. These are `mlist_to_hlist`, `hpack_quality`, `vpack_quality`, `hyphenate`, `linebreak_filter`, `buildpage_filter` and `build_page_insert`. Register `nil` at the callback to pop a function off the stack.

Finally, for the `simple` callbacks `uselanguage`, `contribute_filter`, `pre_dump`, `wrapup_run`, `finish_pdffile`, `finish_pdfpage`, `insert_local_par`, `ligaturing`, `kerning` and `process_rule`. all registered functions are called in order with the same arguments.

Two callbacks are new: the new `mlist_to_mlist` callback is called before `mlist_to_hlist` and should not convert noads to nodes, while the `uselanguage` callback is called from `\uselanguage`.

If you create a new callback with a number for a name, that callback will replace the `process_rule` callback when its number matches the index property of the rule.

PDF resources

This package can perform sophisticated pdf resource management, assigning to every page a resource object containing only the resources referenced on that page. pdf resources are `ExtGstate`, `ColorSpace`, `Pattern` and `Shading` objects that have to be referenced by name (`/name`) instead of with the usual object references (`n 0 R`).* Currently, the only other package managing pdf resources for plain tex is tikz/pgf, and the latter does so by collecting all resources in a single (global) resource object. That approach is not wrong per se, but may cause other tools processing the resulting pdf to retain unneeded resources. Both implementations can safely be used together, but since pgf does not keep track of actual resource use, any resources defined through pgf will be added to the resource dictionary of every subsequent page.

The resource management is implemented in minim-pdfresources.lua and minim-pdfresources.tex, of which the tex file currently only includes pgf compatibility code (and may thus be omitted in the absence of pgf). In the following, `R = require 'minim-pdfresources'` is understood.

You must register resources before you use them. This can be done with `R.add_resource(kind, name, resource)`, where `kind` is one of the resource types, `name` is the name you want to use for it (without a preceding slash) and `resource` is a table that may contain any data you want to associate with the resource. In the `resource` table, either the key `value` must be present (containing the (raw) contents of the resource; will be written to the pdf as-is) or the key `write` (which must be a function that will be called once, to generate the `value`). Registered resources can be retrieved again with `R.get_resource(kind, name)`.

You can refer to registered resources with the `name` you used to register them. Whenever you do so, however, you must mark the reference with a special `late_lua` node that will tell minim to include the resource in the resource list for the page it appears on. These nodes can be created from lua with `R.use_resource_node(kind, name)` or directly inserted by tex with `\withpdfresource {kind} {name}` (the braces are optional).

* This is of course also the case for `Font` resources, but those are already managed by luatex's pdf backend.

Programming helpers

Optional keyword arguments to tex macros can be defined with help of `M.options_scanner()`. An example from the definition of minim-pdf's `\outline`:

```
local s = M.options_scanner()
    :keyword('open', 'closed')
    :string('dest')
    :scan()
s.title = token.scan_string()
M.add_bookmark(s)
```

Here, the `keyword` function adds two opposite keywords: if one is present, its value will be set to `true` and the other's to `false` (the second keyword is optional). The `string` function stores the result of `token.scan_string` under its key. Of the same form you have `int`, `glue`, `dimen`, `argument`, `word` and `list`. All these take an optional second argument: if `true` then the keyword can be repeated and its values will be stored as a list.

The `scan` function, finally, scans all keywords, which may appear in any order. You can give it a table with default values. In the example given above, the argument `s` to `M.add_bookmark` will consist of a table with at most the following entries: `open`, `closed`, `dest` and `title`, though entries whose keywords do not occur will not be present.

This function is particularly useful when used together with `M.luadef('csname', function, ...)`, which defines primitive-like tex macros from lua. There, `function` can be any function (it will be assigned a lua function register) and at the place of the dots you may append `'protected'` and/or `'global'`.

Miscellaneous functions

The small functions `M.msg(...)`, `M.log(...)` and `M.err(...)` include a call to `M.string.format`; additionally, `M.amsg(...)` and `M.alog(...)` do not start a new line.

Both `M.unset` and `\unset` contain the value `-0x7FFFFFFF` that can be used for clearing attributes.

When writing data to pdf literals, `M.pdf_string(...)`, `M.pdf_name(...)` and `M.pdf_date(...)` may be useful: they produce strings that can be written to the pdf directly. Surrounding `<>` or `()` characters or a leading `/` will be included in the return value. The `M.pdf_date` function expects a value of the form `yyyy[-mm[-dd]]` and returns a date of the form `(D:yyyymmdd)`. The function `M.pdf_string` is also available to tex through the macro `\pdfstring`.

Finally the function `M.table_to_text(table)` may be useful when debugging lua code: it dumps a table as a (lua-readable) string. Cyclic references will spin in into an eternal loop, however.

Miscellaneous helper macros

On the tex side, `\voidbox`, `\ignore`, `\spacechar`, `\unbrace`, `\firstoftwo` and `\secondoftwo` should be self-explanatory and uncontroversial additions. `\Ucharcat` works as in xetex. For looking ahead, you can use `\nextif \token {executed if present} {executed if not}` or its siblings `\nextifx` and `\nextifcat`. For defining macros with optional arguments, `\withoptions[default]{code}` will ensure something within square brackets follows `code`.

Finally, `\splitcommalist {code} {list}` will apply `code` to every nonempty item on a comma-separated `list`. Items of the list will be re-tokenised and have surrounding spaces removed. This macro is fully expandable.

Because of their usefulness and simplicity, these macros have been made available without special characters in their names; I hope you can tolerate their presence. Please let me know if their names clash with something important.