The luamplib package

Hans Hagen, Taco Hoekwater and Elie Roux
elie.roux@telecom-bretagne.eu

2009/03/09 v1.01

Abstract

Package to have metapost code typeset directly in a document with LuaTeX.

1 Documentation

This packages aims at providing a simple way to typeset directly metapost code in a document with LuaTeX. LuaTeX is built with the lua `mplib` library, that runs metapost code. This package is basically a wrapper (in Lua) for the Lua `mplib` functions and some TeX functions to have the output of the `mplib` functions in the pdf.

The package need to be in PDF mode in order to output something, as PDF specials are not supported by the DVI format and tools.

The metapost figures are put in a TeX `hbox` with dimensions adjusted to the metapost code.

The code is from the `supp-mpl.lua` and `supp-mpl.tex` files from ConTeXt, they have been adapted to LaTeX and Plain by Elie Roux. The changes are:

- a LaTeX environment
- all TeX macros start by `mplib`
- use of the luatextra printing and module functions

Using this package is easy: in Plain, type your metapost code between the macros `mplibcode` and `endmplibcode`, and in LaTeX in the `mplibcode` environment.

In order to use metapost, some `.mem` files are needed. These files must be generated with the same version of `mplib` as the version of LuaTeX. These files names can be changed, they are by default `luatex-plain.mem` and `luatex-mpfun.mem`. If this package is to be included in a distribution, some values may have to be changed in the file `luamplib.lua`, see comments.

If your distribution does not provide valid `.mem` files (TeXLive 2009 will be the first), you'll have to generate and install them by hand, with the script `luamplib-createmem.lua` included in this package.

2 Files

This package contains three files:

- `luamplib.lua` containing the lua code that calls mplib
- `luamplib.sty` containing the macros for LaTeX and Plain
- `create-mem.lua`, a standalone mem generation script

2.1 `luamplib.lua`

First the `luamplib` module is registered as a LuaTeX module, with some informations. Here we can't name it `mplib`, as the name is already taken.

```
1
2 luamplib = { }
3
4 luamplib.module = {
5     name          = "luamplib",
6     version       =  1.01,
7     date          = "2009/03/09",
8     description   = "Lua functions to typeset Metapost directly with MPLib.",
9     author        = "Hans Hagen, Taco Hoekwater & Elie Roux",
10    copyright     = "ConTeXt Development Team & Elie Roux",
11    license       = "CC0",
12 }
13
14 luatextra.provides_module(luamplib.module)
15
```

This module is a stripped down version of libraries that are used by ConTeXt.

```
16
17 local format, concat, abs = string.format, table.concat, math.abs
18
```

The `mem` file and the format name are hardcoded, and they can be set with TeX if it's useful. The TeX distributions should change these values if necessary.

```
19
20 luamplib.currentformat = "plain"
21 luamplib.currentmem = "luatex-plain.mem"
22
23 local currentformat = luamplib.currentformat
24 local currentmem = luamplib.currentmem
25
26 function luamplib.setformat (name)
27     luamplib.currentformat = name
28 end
29
30 function luamplib.setmemfile(name)
31     luamplib.currentmem = name
```

```
end
```

We use the `kpse` library unless a finder is already defined. To find the .mem files with kpse, we have to set the `engine` environment variable to `metapost`.

```
luamplib.finder = luamplib.finder or function(name, mode, ftype)
    if mode == "w" then
        return name
    else
        local result
        if ftype == 'mem' then
            local envsave = os.getenv('engine')
            os.setenv('engine', 'metapost')
            result = kpse.find_file(name,ftype)
            os.setenv('engine', envsave)
        else
            result = kpse.find_file(name,ftype)
        end
        return result
    end
end

function luamplib.info (...)
    luatextra.module_info('luamplib', format(...))
end

function luamplib.log (...)
    luatextra.module_log('luamplib', format(...))
end

function luamplib.term (...)
    luatextra.module_term('luamplib', format(...))
end

function luamplib.warning (...)
    luatextra.module_warning('luamplib', format(...))
end

function luamplib.error (...)
    luatextra.module_error('luamplib', format(...))
end

```

This is a small hack for LaTeX. In LaTeX we read the metapost code line by line, but it needs to be passed entirely to `luamplib.process`, so we simply add the lines in `luamplib.data` and at the end we call `luamplib.process` on `luamplib.data`.

```
luamplib.data = ""

```

```
function luamplib.resetdata()
    luamplib.data = ""
end

function luamplib.addline(line)
    luamplib.data = luamplib.data .. '\n' .. line
end

function luamplib.processlines()
    luamplib.process(luamplib.data)
    luamplib.resetdata()
end



```

`luamplib.load` This function is the one loading the metapost format we want. It uses the `luamplib.currentformat` and `luamplib.currentmem` to determine the format and the mem file to use.

The rest of this module is not documented. More info can be found in the LuaTEX manual, articles in user group journals and the files that ship with ConTEXt.

```

function luamplib.load()
    local mpx = mplib.new {
        ini_version = false,
        mem_name = currentmem,
        find_file = luamplib.finder
    }
    if mpx then
        luamplib.log("using mem file %s", luamplib.finder(currentmem, 'r', 'mem'))
        return mpx, nil
    else
        return nil, { status = 99, error = "out of memory or invalid format" }
    end
end


function luamplib.report(result)
    if not result then
        luamplib.error("no result object")
    elseif result.status > 0 then
        local t, e, l, f = result.term, result.error, result.log
        if l then
            luamplib.log(l)
        end
        if t then
            luamplib.term(t)
```

```
116         end
117         if e then
118             if result.status == 1 then
119                 luamplib.warning(e)
120             else
121                 luamplib.error(e)
122             end
123         end
124         if not t and not e and not l then
125             if result.status == 1 then
126                 luamplib.warning("unknown error, no error, terminal or log messages, maybe m
127             else
128                 luamplib.error("unknown error, no error, terminal or log messages, maybe mis
129             end
130         end
131     else
132         return true
133     end
134     return false
135 end
136
137 function luamplib.process(data)
138     local converted, result = false, {}
139     local mpx = luamplib.load()
140     if mpx and data then
141         local result = mpx:execute(data)
142         if luamplib.report(result) then
143             if result.fig then
144                 converted = luamplib.convert(result)
145             else
146                 luamplib.warning("no figure output")
147             end
148         end
149     else
150         luamplib.error("Mem file unloadable. Maybe generated with a different version of mpl
151     end
152     return converted, result
153 end
154
155 local function getobjects(result,figure,f)
156     return figure:objects()
157 end
158
159 function luamplib.convert(result, flusher)
160     luamplib.flush(result, flusher)
161     return true -- done
162 end
163
164 local function pdf_startfigure(n,llx,lly,urx,ury)
165     tex.sprint(format("\\mplibstarttoPDF{%s}{%s}{%s}{%s}",llx,lly,urx,ury))
```

```
end

local function pdf_stopfigure()
    tex.sprint("\\mplibstoptoPDF")
end

function pdf_literalcode(fmt,...) -- table
    tex.sprint(format("\\mplibtoPDF{%s}",format(fmt,...)))
end

function pdf_textfigure(font,size,text,width,height,depth)
    text = text:gsub(".","\\hbox{%1}") -- kerning happens in metapost
    tex.sprint(format("\\mplibtextext{%s}{%s}{%s}{%s}{%s}",font,size,text,0,-( 7200/ 7227)/6
end

local bend_tolerance = 131/65536

local rx, sx, sy, ry, tx, ty, divider = 1, 0, 0, 1, 0, 0, 1

local function pen_characteristics(object)
    if luamplib.pen_info then
        local t = luamplib.pen_info(object)
        rx, ry, sx, sy, tx, ty = t.rx, t.ry, t.sx, t.sy, t.tx, t.ty
        divider = sx*sy - rx*ry
        return not (sx==1 and rx==0 and ry==0 and sy==1 and tx==0 and ty==0), t.width
    else
        rx, sx, sy, ry, tx, ty, divider = 1, 0, 0, 1, 0, 0, 1
        return false, 1
    end
end

local function concat(px, py) -- no tx, ty here
    return (sy*px-ry*py)/divider,(sx*py-rx*px)/divider
end

local function curved(ith,pth)
    local d = pth.left_x - ith.right_x
    if abs(ith.right_x - ith.x_coord - d) <= bend_tolerance and abs(pth.x_coord - pth.left_x
        d = pth.left_y - ith.right_y
        if abs(ith.right_y - ith.y_coord - d) <= bend_tolerance and abs(pth.y_coord - pth.le
            return false
        end
    end
    return true
end

local function flushnormalpath(path,open)
    local pth, ith
    for i=1,#path do
        pth = path[i]
```

```
216         if not ith then
217             pdf_literalcode("%f %f m",pth.x_coord,pth.y_coord)
218         elseif curved(ith,pth) then
219             pdf_literalcode("%f %f %f %f %f %f c",ith.right_x,ith.right_y,pth.left_x,pth.lef
220         else
221             pdf_literalcode("%f %f l",pth.x_coord,pth.y_coord)
222         end
223         ith = pth
224     end
225     if not open then
226         local one = path[1]
227         if curved(pth,one) then
228             pdf_literalcode("%f %f %f %f %f %f c",pth.right_x,pth.right_y,one.left_x,one.lef
229         else
230             pdf_literalcode("%f %f l",one.x_coord,one.y_coord)
231         end
232     elseif #path == 1 then
233         -- special case .. draw point
234         local one = path[1]
235         pdf_literalcode("%f %f l",one.x_coord,one.y_coord)
236     end
237     return t
238 end
239
240 local function flushconcatpath(path,open)
241     pdf_literalcode("%f %f %f %f %f %f cm", sx, rx, ry, sy, tx ,ty)
242     local pth, ith
243     for i=1,#path do
244         pth = path[i]
245         if not ith then
246            pdf_literalcode("%f %f m",concat(pth.x_coord,pth.y_coord))
247         elseif curved(ith,pth) then
248             local a, b = concat(ith.right_x,ith.right_y)
249             local c, d = concat(pth.left_x,pth.left_y)
250             pdf_literalcode("%f %f %f %f %f %f c",a,b,c,d,concat(pth.x_coord, pth.y_coord))
251         else
252            pdf_literalcode("%f %f l",concat(pth.x_coord, pth.y_coord))
253         end
254         ith = pth
255     end
256     if not open then
257         local one = path[1]
258         if curved(pth,one) then
259             local a, b = concat(pth.right_x,pth.right_y)
260             local c, d = concat(one.left_x,one.left_y)
261             pdf_literalcode("%f %f %f %f %f %f c",a,b,c,d,concat(one.x_coord, one.y_coord))
262         else
263             pdf_literalcode("%f %f l",concat(one.x_coord,one.y_coord))
264         end
265     elseif #path == 1 then
```

```
        -- special case .. draw point
        local one = path[1]
        pdf_literalcode("%f %f l",concat(one.x_coord,one.y_coord))
    end
    return t
end

```

Support for specials in DVI has been removed.

```

function luamplib.flush(result,flusher)
    if result then
        local figures = result.fig
        if figures then
            for f=1, #figures do
                luamplib.info("flushing figure %s",f)
                local figure = figures[f]
                local objects = getobjects(result,figure,f)
                local fignum = tonumber((figure:filename()):match("([%d]+)$") or figure:char
                local miterlimit, linecap, linejoin, dashed = -1, -1, -1, false
                local bbox = figure:boundingbox()
                local llx, lly, urx, ury = bbox[1], bbox[2], bbox[3], bbox[4] -- faster than
                if urx < llx then
                    -- invalid
                    pdf_startfigure(fignum,0,0,0,0)
                    pdf_stopfigure()
                else
                    pdf_startfigure(fignum,llx,lly,urx,ury)
                    pdf_literalcode("q")
                    if objects then
                        for o=1,#objects do
                            local object = objects[o]
                            local objecttype = object.type
                            if objecttype == "start_bounds" or objecttype == "stop_bounds" t
                                -- skip
                            elseif objecttype == "start_clip" then
                                pdf_literalcode("q")
                                flushnormalpath(object.path,t,false)
                                pdf_literalcode("W n")
                            elseif objecttype == "stop_clip" then
                                pdf_literalcode("Q")
                                miterlimit, linecap, linejoin, dashed = -1, -1, -1, false
                            elseif objecttype == "special" then
                                -- not supported
                            elseif objecttype == "text" then
                                local ot = object.transform -- 3,4,5,6,1,2
                                pdf_literalcode("q %f %f %f %f %f %f cm",ot[3],ot[4],ot[5],o
                                pdf_textfigure(object.font,object.dsize,object.text,object.w
                                pdf_literalcode("Q")
                            else
```

```
local cs = object.color
if cs and #cs > 0 then
    pdf_literalcode(luamplib.colorconverter(cs))
end
local ml = object.miterlimit
if ml and ml ~= miterlimit then
    miterlimit = ml
    pdf_literalcode("%f M",ml)
end
local lj = object.linejoin
if lj and lj ~= linejoin then
    linejoin = lj
    pdf_literalcode("%i j",lj)
end
local lc = object.linecap
if lc and lc ~= linecap then
    linecap = lc
    pdf_literalcode("%i J",lc)
end
local dl = object.dash
if dl then
    local d = format("[%s] %i d",concat(dl.dashes or {}," ")
    if d ~= dashed then
        dashed = d
        pdf_literalcode(dashed)
    end
elseif dashed then
   pdf_literalcode("[] 0 d")
   dashed = false
end
local path = object.path
local transformed, penwidth = false, 1
local open = path and path[1].left_type and path[#path].righ
local pen = object.pen
if pen then
   if pen.type == 'elliptical' then
        transformed, penwidth = pen_characteristics(object)
        pdf_literalcode("%f w",penwidth)
        if objecttype == 'fill' then
            objecttype = 'both'
        end
   else -- calculated by mplib itself
        objecttype = 'fill'
   end
end
if transformed then
    pdf_literalcode("q")
end
if path then
    if transformed then
```

```
                                    flushconcatpath(path,open)
                                else
                                    flushnormalpath(path,open)
                                end
                                if objecttype == "fill" then
                                    pdf_literalcode("h f")
                                elseif objecttype == "outline" then
                                    pdf_literalcode((open and "S") or "h S")
                                elseif objecttype == "both" then
                                    pdf_literalcode("h B")
                                end
                            end
                            if transformed then
                                pdf_literalcode("Q")
                            end
                            local path = object.htap
                            if path then
                                if transformed then
                                    pdf_literalcode("q")
                                end
                                if transformed then
                                    flushconcatpath(path,open)
                                else
                                    flushnormalpath(path,open)
                                end
                                if objecttype == "fill" then
                                    pdf_literalcode("h f")
                                elseif objecttype == "outline" then
                                    pdf_literalcode((open and "S") or "h S")
                                elseif objecttype == "both" then
                                    pdf_literalcode("h B")
                                end
                                if transformed then
                                    pdf_literalcode("Q")
                                end
                            end
                            if cr then
                                pdf_literalcode(cr)
                            end
                        end
                    end
                end
                pdf_literalcode("Q")
                pdf_stopfigure()
            end
        end
    end
end
```

```
function luamplib.colorconverter(cr)
    local n = #cr
    if n == 4 then
        local c, m, y, k = cr[1], cr[2], cr[3], cr[4]
        return format("%.3f %.3f %.3f %.3f k %.3f %.3f %.3f %.3f K",c,m,y,k,c,m,y,k), "0 g 0
    elseif n == 3 then
        local r, g, b = cr[1], cr[2], cr[3]
        return format("%.3f %.3f %.3f rg %.3f %.3f %.3f RG",r,g,b,r,g,b), "0 g 0 G"
    else
        local s = cr[1]
        return format("%.3f g %.3f G",s,s), "0 g 0 G"
    end
end
```

2.2 luamplib.sty

First we need to load fancyvrb, to define the environment mplibcode.

```
\expandafter\ifx\csname ProvidesPackage\endcsname\relax
  \input luatextra.sty
\else
  \NeedsTeXFormat{LaTeX2e}
  \ProvidesPackage{luamplib}
    [2009/03/09 v1.01 mplib package for LuaTeX.]
  \RequirePackage{luatextra}
  \RequirePackage{fancyvrb}
\fi
```

Loading of lua code.

```
\luaUseModule{luamplib}
```

There are (basically) two formats for metapost: *plain* and *mpfun*. The corresponding `.mem` files are (at least will be) `luatex-plain.mem` and `luatex-mpfun.mem` in TEXLive. With these functions you can set the format and the mem files that will be used by this package. Warning: the package never generates the mem files, you have to do it by hand, with `create-mem.lua`.

```
\def\mplibsetformat#1{\directlua0{luamplib.setformat([[#1]])}}

\def\mplibsetmemfile#1{\directlua0{luamplib.setmemfile([[#1]])}}
```

MPLib only works in PDF mode, we don't do anything if we are in DVI mode, and we output a warning.

```
\ifnum\pdfoutput>0
```

```
    \let\mplibtoPDF\pdfliteral
\else
    %\def\MPLIBtoPDF#1{\special{pdf:literal direct #1}} % not ok yet
    \def\mplibtoPDF#1{}
    \expandafter\ifx\csname PackageWarning\endcsname\relax
      \write16{}
      \write16{Warning: MPLib only works in PDF mode, no figure will be output.}
      \write16{}
    \else
      \PackageWarning{mplib}{MPLib only works in PDF mode, no figure will be output.}
    \fi
\fi

```

The Plain-specific stuff.

```

\expandafter\ifx\csname ProvidesPackage\endcsname\relax

\def\mplibsetupcatcodes{
  \catcode`\{=12 \catcode`\}=12 \catcode`\#=12 \catcode`\^=12 \catcode`\~=12
  \catcode`\_=12 \catcode`\%=12 \catcode`\&=12 \catcode`\$=12
}

\def\mplibcode{%
  \bgroup %
  \mplibsetupcatcodes %
  \mplibdocode %
}

\long\def\mplibdocode#1\endmplibcode{%
  \egroup %
  \mplibprocess{#1}%
}

\long\def\mplibprocess#1{%
  \luadirect{luamplib.process([[#1]])}%
}

\else

```

The LaTeX-specific parts. First a Hack for the catcodes in LaTeX.

```

\makeatletter
\begingroup
\catcode`\,=13
\catcode`\-=13
\gdef\FV@hack{%
  \def,{\string,}%
  \def-{\string-}%
```

```
}
\endgroup
```

In LaTeX (it's not the case in plainTeX), we get the metapost code line by line, here is the function handling a line.

```
\newcommand\mplibaddlines[1]{%
  \begingroup %
  \FV@hack %
  \def\FV@ProcessLine##1{%
    \luadirect{luamplib.addline([[##1]])}%
  }%
  \csname FV@SV@#1\endcsname %
  \endgroup %
}

\makeatother
```

The LaTeX environment is a modified `verbatim` environment.

```
\newenvironment{mplibcode}{%
  \VerbatimEnvironment %
  \begin{SaveVerbatim}{memoire}%
}{%
  \end{SaveVerbatim}%
  \mplibaddlines{memoire}%
  \luadirect{luamplib.processlines()}%
}

\fi
```

We use a dedicated scratchbox.

```
\ifx\mplibscratchbox\undefined \newbox\mplibscratchbox \fi
```

We encapsulate the litterals.

```
\def\mplibstarttoPDF#1#2#3#4{
  \hbox\bgroup
  \xdef\MPllx{#1}\xdef\MPlly{#2}%
  \xdef\MPurx{#3}\xdef\MPury{#4}%
  \xdef\MPwidth{\the\dimexpr#3bp-#1bp\relax}%
  \xdef\MPheight{\the\dimexpr#4bp-#2bp\relax}%
  \parskip0pt%
  \leftskip0pt%
  \parindent0pt%
  \everypar{}%
```

```
  \setbox\mplibscratchbox\vbox\bgroup
  \noindent
}

\def\mplibstoptoPDF{%
  \egroup %
  \setbox\mplibscratchbox\hbox %
    {\hskip-\MPllx bp%
     \raise-\MPlly bp%
     \box\mplibscratchbox}%
  \setbox\mplibscratchbox\vbox to \MPheight
    {\vfill
     \hsize\MPwidth
     \wd\mplibscratchbox0pt%
     \ht\mplibscratchbox0pt%
     \dp\mplibscratchbox0pt%
     \box\mplibscratchbox}%
  \wd\mplibscratchbox\MPwidth
  \ht\mplibscratchbox\MPheight
  \box\mplibscratchbox
  \egroup
}

```

Text items have a special handler.

```

\def\mplibtextext#1#2#3#4#5{%
  \begingroup
  \setbox\mplibscratchbox\hbox
    {\font\temp=#1 at #2bp%
     \temp
     #3}%
  \setbox\mplibscratchbox\hbox
    {\hskip#4 bp%
     \raise#5 bp%
     \box\mplibscratchbox}%
  \wd\mplibscratchbox0pt%
  \ht\mplibscratchbox0pt%
  \dp\mplibscratchbox0pt%
  \box\mplibscratchbox
  \endgroup
}

```

2.3 luamplib-createmem.lua

Finally a small standalone file to call with `texlua` that generates `luatex-plain.mem` in the current directory. To generate other formats in other names, simply change the last line. After the `mem` generation, you'll have to install it in a directory

searchable by TEX.

```
577
578 kpse.set_program_name("kpsewhich")
579
580 function finder (name, mode, ftype)
581     if mode == "w" then
582         return name
583     else
584         local result = kpse.find_file(name,ftype)
585         return result
586     end
587 end
588
589 local preamble = [[
590 input %s ; dump ;
591 ]]
592
```

makeformat

```
593
594 makeformat = function (name, mem_name)
595     local mpx = mplib.new {
596         ini_version = true,
597         find_file = finder,
598         job_name =  mem_name,
599         }
600     if mpx then
601         local result
602         result = mpx:execute(string.format(preamble,name))
603         print(string.format("dumping format %s in %s", name, mem_name))
604         mpx:finish()
605     end
606 end
607
608 makeformat("plain", "luatex-plain.mem")
609
```