metafun xl
Hans Hagen
Contents

Introduction 2
1 Text 4
2 Axis 6
3 Outline 8
4 Followtext 11
5 Placeholder 14
6 Arrow 16
7 Shade 19
8 Contour 24
9 Surface 35
10 Mesh 38
11 Function 42
12 Chart 48
13 SVG 57
14 Fonts 59
15 Interface 65
Introduction

For quite a while, around since 1996, the integration of MetaPost into ConTExt became sort of mature but, it took decades of stepwise refinement to reach the state that we’re in now. In this manual I will discuss some of the features that became possible by combining Lua and MetaPost. We already had quite a bit of that for a decade but in 2018, when LuaMetaTEx showed up a next stage was started.

Before we go into details it is good to summarize the steps that were involved in integrating MetaPost and TEx in ConTExt. It indicates a bit what we had and have to deal with which in turn lead to the interfaces we now have.

Originally, TEx had no graphic capabilities: it just needed to know dimensions of the graphics and pass some basic information about what to include to the dvi post processor. So, a MetaPost graphic was normally processed outside the current run, resulting in PostScript graphic, that then had to be included. In pdfTEx there were some more built in options, and therefore the MetaPost code could be processed runtime using some (generic) TEx macros that I wrote. However, that engine still had to launch MetaPost for each graphic, although we could accumulate them and do that between runs. Immediate processing means that we immediately know the dimensions, while a collective run is faster.

In LuaTEx this all changed to very fast runtime processing, made possible because the MetaPost library is embedded in the engine, a decision that we made early in the project and never regret.

With pdfTEx the process was managed by the texexec ConTExt runner but with LuaTEx it stayed under the control of the current run. In the case of pdfTEx the actual embedding was done by TEx macros that interpreted the (relatively simple) PostScript code and turned it into pdf literals. In LuaTEx that job was delegated to Lua.

When using pdfTEx with independent MetaPost runs support for special color spaces, transparency, embedded graphics, outline text, shading and more was implemented using specials and special colors where the color served as reference to some special extension. This works quite well. In LuaTEx the pre- and postscript features, which are properties of picture objects, are used.

In all cases, some information about the current run, for instance layout related information, or color information, has to be passed to the rather isolated MetaPost run. In the case if LuaTEx (and MkIV) the advantage is that processing optional text happens in the same process so there we don’t need to pass information about for instance the current font setup.

In LuaTEx the MetaPost library has a runscript feature, which will call Lua with the given code. This permitted a better integration: we could now ask for specific information (to the TEx end) instead of passing it from the TEx end with each run. In LuaMetaTEx another feature was added: access to the scanners from the Lua end. Although we could already fetch some variables when in Lua this made it possible to extend the MetaPost language in ways not possible before.

Already for a while Alan Braslau and I were working on some new MetaFun code that exploits all these new features. When the scanners came available I sat down and started working on new interfaces and in this manual I will discuss some of these. Some of them are illustrative, others are probably rather useful. The core of what we could call LuaMetaFun (or MetaFun XL when we use the file extension as indicator) is a key-value interface as we have at the TEx end. This interface relates to ConTExt Imtx development and therefore related files have a different suffix: mpxl. However, keep in mind that some are just wrappers around regular MetaPost code so you have the full power of traditional MetaPost at hand.
We can never satisfy all needs, so to some extent this manual also demonstrates how to roll out your own code, but for that you also need to peek into the MetaFun source code too. It will take a while for this manual to complete. I also expect other users to come up with solutions, so maybe in the end we will have a collection of modules for specific tasks.

Hans Hagen
Hasselt NL
August 2019 (and beyond)
1 Text

The MetaFun textext command normally can do the job of typesetting a text snippet quite well.

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw textext("\bf This is text A") withcolor "white";
\stopMPcode

We get:

This is text A

You can use regular ConTeXt commands, so this is valid:

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw textext("\bf This is text A") withcolor "white";
\stopMPcode

Of course you can as well draw a frame in MetaPost but the \framed command has more options, like alignments.

This is text A

Here is a variant using the MetaFun interface:

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw lmt_text [text = "This is text A", color = "white", style = "bold" ];
\stopMPcode

The outcome is more or less the same:

This is text A

Here is another example. The format option is actually why this command is provided.

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw lmt_text [text = decimal 123.45678, color = "white",

\stopMPcode

Here is another example. The format option is actually why this command is provided.

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw lmt_text [text = decimal 123.45678, color = "white",

\stopMPcode

Here is another example. The format option is actually why this command is provided.

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkred";
draw lmt_text [text = decimal 123.45678, color = "white",

\stopMPcode
The following parameters can be set:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>numeric</td>
<td>0</td>
<td>adapts the dimensions to the font (yes uses the default strut)</td>
</tr>
<tr>
<td>strut</td>
<td>string</td>
<td>auto</td>
<td>one of these lft, urt like anchors</td>
</tr>
<tr>
<td>style</td>
<td>string</td>
<td></td>
<td>a format specifier using @ instead of a percent sign</td>
</tr>
<tr>
<td>color</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>anchor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>position</td>
<td>pair</td>
<td>origin</td>
<td></td>
</tr>
<tr>
<td>trace</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

The next example demonstrates the positioning options:

\startMPcode
fill fullcircle xyscaled (8cm,1cm) withcolor "darkblue" ;
fill fullcircle scaled .5mm withcolor "white" ;
draw lmt_text [ 
  text = "left", 
  color = "white", 
  style = "bold", 
  anchor = "lft", 
  position = (-1mm,2mm),
];
draw lmt_text [ 
  text = "right", 
  color = "white", 
  style = "bold", 
  anchor = "rt", 
  offset = 3mm, 
];
\stopMPcode
2 Axis

The axis macro is the result of one of the first experiments with the key/value interface in MetaFun. Let’s show a lot in one example:

\startMPcode
draw lmt_axis [
    sx = 5mm, sy = 5mm,
    nx = 20, ny = 10,
    dx = 5, dy = 2,
    tx = 10, ty = 10,

    list = {
        [ 
            connect = true,
            color = "darkred",
            close = true,
            points = { (1, 1), (15, 8), (2, 10) },
            texts = { "segment 1", "segment 2", "segment 3" }
        ],
        [ 
            connect = true,
            color = "darkgreen",
            points = { (2, 2), (4, 1), (10, 3), (16, 8), (19, 2) },
            labels = { "a", "b", "c", "d", "e" }
        ],
        [ 
            connect = true,
            color = "darkblue",
            close = true,
            points = { (5, 3), (8, 8), (16, 1) },
            labels = { "1", "2", "3" }
        ]
    }
],

] withpen pencircle scaled 1mm ;
\stopMPcode

This macro will probably be extended at some point.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dx</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>tx</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sx</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>startx</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ny</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dy</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.1

```
ty numeric 0
sy numeric 1
starty numeric 0

samples list
list list
connect boolean false
list list
close boolean false
samplecolors list
axiscolor string
textcolor string
```
3 Outline

In a regular text you can have outline characters by setting a (pseudo) font feature but sometimes you want to play a bit more with this. In MetaFun we always had that option. In MkII we call pstoedit to turn text into outlines, in MkIV we do that by manipulating the shapes directly. And, as with some other extensions, in lmtx a new interface has been added, but the underlying code is the same as in MkIV.

In figure 3.1 we see two examples:

\startMPcode{doublefun}
  draw lmt_outline [  
    text    = "hello",  
    kind    = "draw",  
    drawcolor = "darkblue",  
  ] xsized .45TextWidth ;
\stopMPcode

\startMPcode{doublefun}
  draw lmt_outline [  
    text    = "hello",  
    kind    = "both",  
    fillcolor = "middlegray",  
    drawcolor = "darkgreen",  
    rulethickness = 1/5,  
  ] xsized .45TextWidth ;
\stopMPcode

\begin{figure}[h]
\includegraphics[width=\textwidth]{figure3.1}
\caption{Drawing and/or filling an outline.}
\end{figure}

Normally the fill ends up below the draw but we can reverse the order, as in figure 3.2, where we coded the leftmost example as:

\startMPcode{doublefun}
  draw lmt_outline [  
    text    = "hello",  
    kind    = "reverse",  
    fillcolor = "darkred",  
    drawcolor = "darkblue",  
    rulethickness = 1/2,  
  ] xsized .45TextWidth ;
\stopMPcode
It is possible to fill and draw in one operation, in which case the same color is used for both, see figure 3.3 for an example of this. This is a low level optimization where the shape is only output once.

This interface is much nicer than the one where each variant (the parameter \texttt{kind} above) had its own macro due to the need to group properties of the outline and fill. Let’s show some more:

\begin{verbatim}
\startMPcode{doublefun}
\draw lmtoutline [
  text = "\obeydiscretionaries\samplefile{tufte}",
  align = "normal",
  kind = "draw",
  drawcolor = "darkblue",
] xsized TextWidth ;
\stopMPcode
\end{verbatim}

In this case we feed the text into the \texttt{\textbf{framed}} macro so that we get a properly aligned paragraph of text, as demonstrated in figure 3.4 and ??.

\begin{verbatim}
\startMPcode{doublefun}
\draw lmtoutline [
  text = "\obeydiscretionaries\samplefile{ward}",
  align = "normal,tolerant",
  style = "bold",
] xsized TextWidth ;
\stopMPcode
\end{verbatim}

We thrive in information—thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

\begin{figure}
\centering
\begin{verbatim}
\startMPcode{doublefun}
\draw lmtoutline [
  text = "\obeydiscretionaries\samplefile{tufte}",
  align = "normal",
  kind = "draw",
  drawcolor = "darkblue",
] xsized TextWidth ;
\stopMPcode
\end{verbatim}
\caption{Outlining a paragraph of text.}
\end{figure}

Outsourcing uncorrected draft
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**Figure 3.5** Outlining a paragraph of text with a specific width.

We summarize the parameters:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kind</td>
<td>string</td>
<td>draw</td>
<td>One of draw, fill, both, reverse and fillup.</td>
</tr>
<tr>
<td>fillcolor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>drawcolor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rulethickness</td>
<td>numeric</td>
<td>1/10</td>
<td></td>
</tr>
<tr>
<td>align</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>style</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4 Followtext

Typesetting text along a path started as a demo if communication between \TeX{} and MetaPost in the early days of MetaFun. In the meantime the implementation has been modernized a few times and the current implementation feels okay, especially now that we have a better user interface. Here is an example:

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_followtext [ \\
    text = "How well does it work {\bf 1}! ", \\
    path = fullcircle scaled 4cm, \\
    trace = true, \\
    spread = true, \\
  ] sized 5cm ; \\
\stopMPcode
\end{verbatim}

Here is the same example but with the text in the reverse order. The results of both examples are shown in figure 4.1.

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_followtext [ \\
    text = "How well does it work {\bf 2}! ", \\
    path = fullcircle scaled 4cm, \\
    trace = true, \\
    spread = false, \\
    reverse = true, \\
  ] sized 5cm ; \\
\stopMPcode
\end{verbatim}

There are not that many options. One is autoscale which makes the shape and text match. Figure 4.2 shows what happens.

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_followtext [ \\
    text = "How well does it work {\bf 3}! ", \\
    trace = true, \\
  ] sized 5cm ; \\
\stopMPcode
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.1.png}
\caption{Figure 4.1}
\end{figure}
autoscaleup = "yes"
] sized 5cm ;
\stopMPcode
\startMPcode{doublefun}
  draw lmt_followtext [
    text = "How well does it work \bf 4!",
    path = fullcircle scaled 2cm,
    trace = true,
    autoscaleup = "max"
  ] sized 5cm ;
\stopMPcode

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{path.png}
\caption{Figure 4.2}
\end{figure}

You can use quite strange paths, like the one show in figure 4.3. Watch the parenthesis around the path. this is really needed in order for the scanner to pick up the path (otherwise it sees a pair).

\startMPcode{doublefun}
  draw lmt_followtext [
    text = "\samplefile {zapf}",
    path = ((3,0) .. (1,0) .. (5,0) .. (2,0) .. (4,0) .. (3,0)),
    autoscaleup = "max"
  ] sized TextWidth ;
\stopMPcode

The small set of options is:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spread</td>
<td>string</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>trace</td>
<td>numeric</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>numeric</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>autoscaleup</td>
<td>numeric</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>autoscaledown</td>
<td>string</td>
<td>no</td>
<td>(fullcircle)</td>
</tr>
<tr>
<td>path</td>
<td>string</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Followtext uncorrected draft
5 Placeholder

Placeholders are an old ConTeXt features and have been around since we started using MetaPost. They are used as dummy figure, just in case one is not (yet) present. They are normally activated by loading a MetaFun library:

\useMPLibrary[dum]

Just because it could be done conveniently, placeholders are now defined at the MetaPost end instead of as useable MetaPost graphic at the \TeX end. The variants and options are demonstrated using side floats.

\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "red",
  alternative = "circle".
] ;
\stopMPcode

\begin{figure}[h]
\centering
\scalebox{0.4}{\begin{verbatim}
\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "red",
  alternative = "circle".
] ;
\stopMPcode
\end{verbatim}}
\caption{Figure 5.1}
\end{figure}

In addition to the traditional random circle we now also provide rectangles and triangles. Maybe some day more variants will show up.

\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "green",
  alternative = "square".
] ;
\stopMPcode

\begin{figure}[h]
\centering
\scalebox{0.4}{\begin{verbatim}
\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "green",
  alternative = "square".
] ;
\stopMPcode
\end{verbatim}}
\caption{Figure 5.2}
\end{figure}

Here we set the colors but in the image placeholder mechanism we cycle through colors automatically. Here we use primary, rather dark, colors.

\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "blue",
  alternative = "triangle".
] ;
\stopMPcode

\begin{figure}[h]
\centering
\scalebox{0.4}{\begin{verbatim}
\startMPcode
lmt_placeholder [
  width = 4cm,
  height = 3cm,
  color = "blue",
  alternative = "triangle".
] ;
\stopMPcode
\end{verbatim}}
\caption{Figure 5.3}
\end{figure}

If you want less dark colors, the reduction parameter can be used to interpolate between the given color and white; its value is therefore a value between zero (default) and 1 (rather pointless as it produces white).
We demonstrate this with four variants, all circles. Of course you can also use lighter colors, but this option was needed for the image placeholders anyway.

\startMPcode
lmt_placeholder [ width = 4\text{cm},
height = 3\text{cm},
color = "yellow",
alternative = "circle",
reduction = 0.25,
];
\stopMPcode

**Figure 5.4**

There are only a few possible parameters. As you can see, proper dimensions need to be given because the defaults are pretty small.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>string</td>
<td>red</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>reduction</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>alternative</td>
<td>string</td>
<td>circle</td>
<td></td>
</tr>
</tbody>
</table>
6 Arrow

Arrows are somewhat complicated because they follow the path, are constructed using a pen, have a fill and draw, and need to scale. One problem is that the size depends on the pen but the pen normally is only known afterwards.

To some extent MetaFun can help you with this issue. In figure 6.1 we see some variants. The definitions are given below:

\startMPcode
draw lmt_arrow [  
    path = (fullcircle scaled 3cm),  
    withpen pencircle scaled 2mm  
    withcolor "darkred" ;  
\stopMPcode

\startMPcode
draw lmt_arrow [  
    path = (fullcircle scaled 3cm),  
    length = 8,  
]  
    withpen pencircle scaled 2mm  
    withcolor "darkgreen" ;  
\stopMPcode

\startMPcode
draw lmt_arrow [  
    path = (fullcircle scaled 3cm rotated 45),  
    pen = (pencircle xscaled 2mm yscaled 1mm rotated 45),  
]  
    withpen pencircle xscaled 2mm yscaled 1mm rotated 45  
    withcolor "darkblue" ;  
\stopMPcode

\startMPcode
pickup pencircle xscaled 2mm yscaled 1mm rotated 45 ;  
draw lmt_arrow [  
    path = (fullcircle scaled 3cm rotated 45),  
    pen = "auto",  
]  
    withcolor "darkyellow" ;  
\stopMPcode

There are some options that influence the shape of the arrowhead and its location on the path. You can for instance ask for two arrowheads:

\startMPcode
    pickup pencircle scaled 1mm ;
\stopMPcode
Figure 6.1

draw lmt_arrow [
    pen = "auto",
    location = "both"
    path = fullcircle scaled 3cm rotated 90,
] withcolor "darkgreen" ;
\stopMPcode

The shape can also be influenced although often this is not that visible:
\startMPcode
    pickup pencircle scaled 1mm ;
    draw lmt_arrow [
        kind = "draw",
        pen = "auto",
        penscale = 4,
        location = "middle",
        alternative = "curved",
        path = fullcircle scaled 3cm,
    ] withcolor "darkblue" ;
\stopMPcode

The location can also be given as percentage, as this example demonstrates. Watch how we draw only arrow heads:
\startMPcode
    pickup pencircle scaled 1mm ;
\begin{verbatim}
for i = 0 step 5 until 100 :
    draw lmt_arrow [
        alternative = "dimpled",
        pen = "auto",
        location = "percentage",
        percentage = i,
        dimple = (1/5 + i/200),
        headonly = (i = 0),
        path = fullcircle scaled 3cm,
    ] withcolor "darkyellow";
endfor;
\end{verbatim}

The supported parameters are:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pen</td>
<td>path</td>
<td>auto</td>
<td></td>
</tr>
<tr>
<td>kind</td>
<td>string</td>
<td>fill</td>
<td>fill or draw</td>
</tr>
<tr>
<td>dimple</td>
<td>numeric</td>
<td>1/5</td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td>numeric</td>
<td>3/4</td>
<td></td>
</tr>
<tr>
<td>penscale</td>
<td>numeric</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>numeric</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>angle</td>
<td>numeric</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>string</td>
<td>end</td>
<td>end, middle or both</td>
</tr>
<tr>
<td>alternative</td>
<td>string</td>
<td>normal</td>
<td>normal, dimpled or curved</td>
</tr>
<tr>
<td>percentage</td>
<td>numeric</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>headonly</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
7 Shade

This interface is still experimental!

Shading is complex. We go from one color to another on a continuum either linear or circular. We have to make sure that we cover the whole shape and that means that we have to guess a little, although one can influence this with parameters. It can involve a bit of trial and error, which is more complex than using a graphical user interface but this is the price we pay. It goes like this:

\startMPcode
definecolor [ name = "MyColor3", r = 0.22, g = 0.44, b = 0.66 ];
definecolor [ name = "MyColor4", r = 0.66, g = 0.44, b = 0.22 ];
draw lmt_shade [  
  path = fullcircle scaled 4cm,  
  direction = "right",  
  domain = { 0, 2 },  
  colors = { "MyColor3", "MyColor4" },
];
draw lmt_shade [  
  path = fullcircle scaled 3cm,  
  direction = "left",  
  domain = { 0, 2 },  
  colors = { "MyColor3", "MyColor4" },
] shifted (45mm,0);
draw lmt_shade [  
  path = fullcircle scaled 5cm,  
  direction = "up",  
  domain = { 0, 2 },  
  colors = { "MyColor3", "MyColor4" },
] shifted (95mm,0);
draw lmt_shade [  
  path = fullcircle scaled 1cm,  
  direction = "down",  
  domain = { 0, 2 },  
  colors = { "MyColor3", "MyColor4" },
] shifted (135mm,0);
\stopMPcode

Normally this is good enough as demonstrated in figure 7.1 because we use shades as backgrounds. In the case of a circular shade we need to tweak the domain because guessing doesn’t work well.

\startMPcode

draw lmt_shade [  
  path = fullsquare scaled 4cm,  
  alternative = "linear",
] shifted (135mm,0);
\stopMPcode
Figure 7.1  Simple circular shades.

```plaintext
direction  = "right",
colors     = { "MyColor3", "MyColor4" },
]
);
draw lmt_shade [  
    path      = fullsquare scaled 3cm,  
    direction = "left",  
    alternative = "linear",  
    colors     = { "MyColor3", "MyColor4" },
]
shifted (45mm,0) ;

draw lmt_shade [  
    path      = fullsquare scaled 5cm,  
    direction = "up",  
    alternative = "linear",  
    colors     = { "MyColor3", "MyColor4" },
]
shifted (95mm,0) ;

draw lmt_shade [  
    path      = fullsquare scaled 1cm,  
    direction = "down",  
    alternative = "linear",  
    colors     = { "MyColor3", "MyColor4" },
]
shifted (135mm,0) ;
\stopMPcode
```

Figure 7.2  Simple rectangular shades.
The direction relates to the bounding box. Instead of a keyword you can also give two values, indicating points on the bounding box. Because a bounding box has four points, the up direction is equivalent to \[ \{0.5, 2.5\} \].

The parameters center, factor, vector and domain are a bit confusing but at some point the way they were implemented made sense, so we keep them as they are. The center moves the center of the path that is used as anchor for one color proportionally to the bounding box: the given factor is multiplied by half the width and height.

\begin{verbatim}
\startMPcode
draw lmt_shade [ 
    path = fullcircle scaled 5cm, 
    domain = \{.2, 1.6\}, 
    center = \{1/10, 1/10\}, 
    direction = "right", 
    colors = \{"MyColor3", "MyColor4"\}, 
    trace = true, 
] ;
\stopMPcode
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.3}
\caption{Moving the centers.}
\end{figure}

A vector takes the given points on the path as centers for the colors, see figure 7.4.

\begin{verbatim}
\startMPcode
draw lmt_shade [ 
    path = fullcircle scaled 5cm, 
    domain = \{.2, 1.6\}, 
    vector = \{2, 4\}, 
    direction = "right", 
    colors = \{"MyColor3", "MyColor4"\}, 
    trace = true, 
] ;
\stopMPcode
\end{verbatim}

Messing with the radius in combination with the previously mentioned domain is really trial and error, as seen in figure 7.5.

\begin{verbatim}
\startMPcode
draw lmt_shade [ 
\end{verbatim}
Figure 7.4 Using a vector (points).

\begin{verbatim}
path = fullcircle scaled 5cm,
domain = { 0.5, 2.5 },
radius = { 2cm, 6cm },
direction = "right",
colors = { "MyColor3", "MyColor4" },
trace = true,
\end{verbatim}

Figure 7.5 Tweaking the radius.

But actually the radius used alone works quite well as shown in figure 7.6.

\begin{verbatim}
\startMPcode
draw lmt_shade [ 
   path = fullcircle scaled 5cm,
   colors = { "red", "green" },
   trace = true,
] ;
\stopMPcode
\begin{verbatim}
draw lmt_shade [ 
   path = fullcircle scaled 5cm,
   colors = { "red", "green" },
   radius = 2.5cm,
] ;
\end{verbatim}
trace = true,
] shifted (6cm,0);

draw lmt_shade [
  path = fullcircle scaled 5cm,
  colors = { "red", "green" },
  radius = 2.0cm,
  trace = true,
] shifted (12cm,0);
\stopMPcode

![Figure 7.6](image_url) Just using the radius.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternative</td>
<td>string</td>
<td>circular or linear</td>
<td></td>
</tr>
<tr>
<td>path</td>
<td>path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trace</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>domain</td>
<td>set of numerics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>radius</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>origin</td>
<td>pair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td>set of numerics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>colors</td>
<td>set of strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>center</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td>string</td>
<td>up, down, left, right</td>
<td>two points on the boundingbox</td>
</tr>
<tr>
<td></td>
<td>set of numerics</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8 Contour

This feature started out as experiment triggered by a request on the mailing list. In the end it was a nice exploration of what is possible with a bit of Lua. In a sense it is more subsystem than a simple MetaPost macro because quite some Lua code is involved and more might be used in the future. It’s part of the fun.

A contour is a line through equivalent values $z$ that result from applying a function to two variables $x$ and $y$. There is quite a bit of analysis needed to get these lines. In MetaFun we currently support three methods for generating a colorful background and three for putting lines on top:

One solution is to use the isolines and isobands methods are described on the marching squares page of wikipedia:

https://en.wikipedia.org/wiki/Marching_squares

This method is relative efficient as we don’t do much optimization, simply because it takes time and the gain is not that much relevant. Because we support filling of multiple curves in one go, we get efficient paths anyway without side effects that normally can occur from many small paths alongside. In these days of multi megabyte movies and sound clips a request of making a pdf file small is kind of strange anyway. In practice the penalty is not that large.

As background we can use a bitmap. This method is also quite efficient because we use indexed colors which results in a very good compression. We use a simple mapping on a range of values.

A third method is derived from the one that is distributed as C source file at:

https://physiology.arizona.edu/people/secomb/contours
https://github.com/secomb/GreensV4

We can create a background image, which uses a sequence of closed curves\(^1\). It can also provide two variants of lines around the contours (we tag them shape and shade). It’s all a matter of taste. In the meantime I managed to optimize the code a bit and I suppose that when I buy a new computer (the code was developed on an 8 year old machine) performance is probably acceptable.

In order of useability you can think of isoband (band) with isolines (cell), bitmap (bitmap) with isolines (cell) and finally shapes (shape) with edges (edge). But let’s start with a couple of examples.

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_contour [  
    xmin = 0, xmax = 4*pi, xstep = .05,  
    ymin = -6, ymax = 6, ystep = .05,  
    levels = 7,  
    height = 5cm,  
    preamble = "local sin, cos = math.sin, math.cos",  
    function = "cos(x) + sin(y)",  
    background = "bitmap",  
  ]
\end{verbatim}

\(^1\) I have to figure out how to improve it a bit so that multiple path don’t get connected.
In figure 8.1 we see the result. There is a in this case black and white image generated and on top of that we see lines. The step determines the resolution of the image. In practice using a bitmap is quite okay and also rather efficient: we use an indexed colorspace and, as already was mentioned, because the number of colors is limited such an image compresses well. A different rendering is seen in figure 8.2 where we use the shape method for the background. That method creates outlines but is much slower, and when you use a high resolution (small step) it can take quite a while to identify the shapes. This is why we set the cache flag.

```
\startMPcode
{doublefun}

draw lmt_contour [ 
  xmin = 0, xmax = 4*pi, xstep = .10, 
  ymin = -6, ymax = 6, ystep = .10, 

  levels = 7, 
  preamble = "local sin, cos = math.sin, math.cos", 
  function = "cos(x) - sin(y)", 
  background = "shape", 
  foreground = "shape", 
  linewidth = 1/2, 
  cache = true,
]

\stopMPcode
```

We mentioned colorspace but haven’t seen any color yet, so let’s set some in figure 8.3. Two variants are shown: a background shape with foreground shape and a background bitmap with a foreground edge. The bitmap renders quite fast, definitely when we compare with the shape, while the quality is as good at this size.
\textbf{Figure 8.2}

\begin{verbatim}
\startMPcode
{doublefun}

\draw lmt_contour[
    xmin = -10, xmax = 10, xstep = .1,
    ymin = -10, ymax = 10, ystep = .1,
    levels = 10,
    height = 7cm,
    color = "shade({1/2,1/2,0},{0,0,1/2})",
    function = "x^2 + y^2",
    background = "shape",
    foreground = "shape",
    linewidth = 1/2,
    cache = true,
] xsized .45TextWidth ;
\stopMPcode
\end{verbatim}

\textbf{Figure 8.3}

We use the doublefun instance because we need to be sure that we don’t run into issues with scaled numbers, the default model in MetaPost. The function that gets passed is not using MetaPost but Lua, so basically you can do very complex things. Here we directly pass code, but you can for instance also do this:
function document.MyContourA(x,y)
    return x^2 + y^2
end

and then function = "document.MyContourA(x,y)". As long as the function returns a valid number we're okay. When you pass code directly you can use the preamble key to set local shortcuts. In the previous examples we took \( \sin \) and \( \cos \) from the math library but you can also roll out your own functions and/or use the more elaborate \( xmath \) library. The color parameter is also a function, one that returns one or three arguments. In the next example we use \( \text{lin} \) to calculate a fraction of the current level and total number of levels.

\[
\text{draw lmt_contour [}
\begin{align*}
\text{xmin} &= -3, \quad \text{xmax} = 3, \quad \text{xstep} = .01, \\
\text{ymin} &= -1, \quad \text{ymax} = 1, \quad \text{ystep} = .01, \\
\text{levels} &= 10, \\
\text{default} &= .5, \\
\text{height} &= 5\text{cm}, \\
\text{function} &= "x^2 + y^2 + x + y/2", \\
\text{color} &= "\text{lin}(l), 0, 1/2", \\
\text{background} &= "\text{bitmap}" \\
\text{foreground} &= "\text{none}" \\
\text{cache} &= \text{true},
\end{align*}
\] xsized TextWidth ;
\]

Instead of a bitmap we can use an isoband, which boils down to a set of tiny shapes that make up a bigger one. This is shown in figure 8.5.

\[
\text{draw lmt_contour [}
\]

\text{Figure 8.4}

x = [-3,3];  y = [-1,1];

\[x^2 + y^2 + x + y/2\]

Contour
You can draw several functions and see where they overlap:

\startMPcode{doublefun}
\draw lmt_contour [
    xmin = -\pi, xmax = 4*\pi, xstep = .1,
    ymin = -3, ymax = 3, ystep = .1,
    range = [0, .1],
    preamble = "local sin, cos = math.sin, math.cos",
    functions = [
        "sin(x) + sin(y)", "sin(x) + cos(y)",
        "cos(x) + sin(y)", "cos(x) + cos(y)"
    ],
    background = "bitmap",
    linecolor = "black",
    linewidth = 1/10,
    color = "shade({1,1,0},{0,0,1})"
]
\stopMPcode
The range determines the \( z \) value(s) that we take into account. You can also pass a list of colors to be used. In figure 8.7 this is demonstrated. There we also show a variant foreground cell, which uses a bit different method for calculating the edges.

\[ \text{\texttt{\textbackslash startMPcode\{doublefun\}}} \]
\begin{verbatim}
    draw lmt_contour [  
        xmin = -2\pi, xmax = 2\pi, xstep = .01,  
        ymin = -3, ymax = 3, ystep = .01,  
        range = [{-.1,.1}],  
        preamble = "local sin, cos = math.sin, math.cos",  
        functions = [{"sin(x) + sin(y)", "sin(x) + cos(y)"}],  
        background = "bitmap",  
        foreground = "cell",  
        linecolor = "white",  
        linewidth = 1/10,  
        colors = [{(1/2,1/2,1/2), red, green, blue}],  
        level = 3,  
        linewidth = 6,  
        cache = true,  
    ] xsized TextWidth ;
\end{verbatim}
\[ \text{\texttt{\textbackslash stopMPcode}} \]

Here the number of levels depends on the number of functions as each can overlap with another; for instance the outcome of two functions can overlap or not which means 3 cases, and with a value not being seen that gives 4 different cases.

\[ \text{\texttt{\textbackslash startMPcode\{doublefun\}}} \]
\begin{verbatim}
    draw lmt_contour [  
        xmin = -2\pi, xmax = 2\pi, xstep = .01,  
        ymin = -3, ymax = 3, ystep = .01,  
        range = [{-.1,.1}],  
        preamble = "local sin, cos = math.sin, math.cos",  
        functions = [{"sin(x) + sin(y)", "sin(x) + cos(y)"}],  
        background = "bitmap",  
        foreground = "cell",  
        linecolor = "white",  
        linewidth = 1/10,  
        colors = [{(1/2,1/2,1/2), red, green, blue}],  
        level = 3,  
        linewidth = 6,  
        cache = true,  
    ] xsized TextWidth ;
\end{verbatim}
\[ \text{\texttt{\textbackslash stopMPcode}} \]

This a bit of a playground: more variants might show up in due time.
Of course one can wonder how useful showing many functions but it can give nice pictures, as shown in figure 8.8.
We can enlarge the window, which is demonstrated in figure 8.9. I suppose that such images only make sense in educational settings.

In figure 8.10 we see different combinations of backgrounds (in color) and foregrounds (edges) in action.
There are quite some settings. Some deal with the background, some with the foreground and quite some deal with the legend.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmin</td>
<td>numeric</td>
<td>0</td>
<td>needs to be set</td>
</tr>
<tr>
<td>xmax</td>
<td>numeric</td>
<td>0</td>
<td>needs to be set</td>
</tr>
<tr>
<td>ymin</td>
<td>numeric</td>
<td>0</td>
<td>needs to be set</td>
</tr>
<tr>
<td>ymax</td>
<td>numeric</td>
<td>0</td>
<td>needs to be set</td>
</tr>
<tr>
<td>xstep</td>
<td>numeric</td>
<td>0</td>
<td>auto 1/200 when zero</td>
</tr>
<tr>
<td>ystep</td>
<td>numeric</td>
<td>0</td>
<td>auto 1/200 when zero</td>
</tr>
<tr>
<td>checkresult</td>
<td>boolean</td>
<td>false</td>
<td>checks for overflow and NaN</td>
</tr>
<tr>
<td>defaultnan</td>
<td>numeric</td>
<td>0</td>
<td>the value to be used when NaN</td>
</tr>
<tr>
<td>defaultinf</td>
<td>numeric</td>
<td>0</td>
<td>the value to be used when overflow</td>
</tr>
<tr>
<td>levels</td>
<td>numeric</td>
<td>10</td>
<td>number of different levels to show</td>
</tr>
<tr>
<td>level</td>
<td>numeric</td>
<td></td>
<td>only show this level (foreground)</td>
</tr>
<tr>
<td>preamble</td>
<td>string</td>
<td></td>
<td>shortcuts</td>
</tr>
<tr>
<td>function</td>
<td>string</td>
<td>(x + y)</td>
<td>the result z value</td>
</tr>
<tr>
<td>functions</td>
<td>list</td>
<td></td>
<td>multiple functions (overlapping levels)</td>
</tr>
<tr>
<td>color</td>
<td>string</td>
<td>(\text{lin}(l))</td>
<td>the result color value for level (l) (1 or 3 values)</td>
</tr>
<tr>
<td>colors</td>
<td>numeric</td>
<td></td>
<td>used when set</td>
</tr>
<tr>
<td>background</td>
<td>string</td>
<td>bitmap</td>
<td>band, bitmap, shape</td>
</tr>
<tr>
<td>foreground</td>
<td>string</td>
<td>auto</td>
<td>cell, edge, shape auto</td>
</tr>
<tr>
<td>linewidth</td>
<td>numeric</td>
<td>.25</td>
<td></td>
</tr>
<tr>
<td>linecolor</td>
<td>string</td>
<td>gray</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>numeric</td>
<td>0</td>
<td>automatic when zero</td>
</tr>
<tr>
<td>height</td>
<td>numeric</td>
<td>0</td>
<td>automatic when zero</td>
</tr>
<tr>
<td>trace</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>legend</td>
<td>string</td>
<td>all</td>
<td>x y z function range all</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>-------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>legendheight</td>
<td>numeric</td>
<td>LineHeight</td>
<td></td>
</tr>
<tr>
<td>legendwidth</td>
<td>numeric</td>
<td>LineHeight</td>
<td></td>
</tr>
<tr>
<td>legendgap</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>legenddistance</td>
<td>numeric</td>
<td>EmWidth</td>
<td></td>
</tr>
<tr>
<td>textdistance</td>
<td>numeric</td>
<td>2EmWidth/3</td>
<td></td>
</tr>
<tr>
<td>functiondistance</td>
<td>numeric</td>
<td>ExHeight</td>
<td></td>
</tr>
<tr>
<td>functionstyle</td>
<td>string</td>
<td>ConTeXt style name</td>
<td></td>
</tr>
<tr>
<td>xformat</td>
<td>string</td>
<td>@0.2N</td>
<td>number format template</td>
</tr>
<tr>
<td>yformat</td>
<td>string</td>
<td>@0.2N</td>
<td>number format template</td>
</tr>
<tr>
<td>zformat</td>
<td>string</td>
<td>@0.2N</td>
<td>number format template</td>
</tr>
<tr>
<td>xstyle</td>
<td>string</td>
<td>ConTeXt style name</td>
<td></td>
</tr>
<tr>
<td>ystyle</td>
<td>string</td>
<td>ConTeXt style name</td>
<td></td>
</tr>
<tr>
<td>zstyle</td>
<td>string</td>
<td>ConTeXt style name</td>
<td></td>
</tr>
<tr>
<td>axisdistance</td>
<td>numeric</td>
<td>ExHeight</td>
<td></td>
</tr>
<tr>
<td>axislinewidth</td>
<td>numeric</td>
<td>.25</td>
<td></td>
</tr>
<tr>
<td>axisoffset</td>
<td>numeric</td>
<td>ExHeight/4</td>
<td></td>
</tr>
<tr>
<td>axiscolor</td>
<td>string</td>
<td>black</td>
<td></td>
</tr>
<tr>
<td>ticklength</td>
<td>numeric</td>
<td>ExHeight</td>
<td></td>
</tr>
<tr>
<td>xtick</td>
<td>numeric</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ytick</td>
<td>numeric</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>xlabel</td>
<td>numeric</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ylabel</td>
<td>numeric</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Figure 8.10
9 Surface

This is work in progress so only some examples are shown here. Yet to be decided is how we deal with axis and such.

In figure 9.1 we see an example of a plot with axis as well as lines drawn.

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_surface [ 
    preamble = "local sin, cos = math.sin, math.cos",
    code = "sin(x*x) - cos(y*y)"
    xmin = -3,
    xmax = 3,
    ymin = -3,
    ymax = 3,
    xvector = { -0.3, -0.3 },
    height = 5cm,
    axis = { 40mm, 40mm, 30mm },
    clipaxis = true,
    axiscolor = "gray",
  ] xsized .8TextWidth ;
\stopMPcode
\end{verbatim}

In figure 9.2 we don't draw the axis and lines. We also use a high resolution.

\begin{verbatim}
\startMPcode{doublefun}
  draw lmt_surface [ 
    preamble = "local sin, cos = math.sin, math.cos",

\end{verbatim}
code = "sin(x*x) - cos(y*y)"
color = "f, f/2, 1-f"
color = "f, f, 0"
xstep = .02,
ystep = .02,
xvector = { -0.4, -0.4 },
height = 5cm,
lines = false,
] xsized .8TextWidth ;
\stopMPcode

Figure 9.2

The preliminary set of parameters is:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>string</td>
<td></td>
<td>color</td>
</tr>
<tr>
<td>linecolor</td>
<td>numeric</td>
<td>1</td>
<td>gray scale</td>
</tr>
<tr>
<td>xmin</td>
<td>numeric</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>xmax</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ymin</td>
<td>numeric</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>ymax</td>
<td>numeric</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>xstep</td>
<td>numeric</td>
<td>.1</td>
<td></td>
</tr>
<tr>
<td>ystep</td>
<td>numeric</td>
<td>.1</td>
<td></td>
</tr>
<tr>
<td>snap</td>
<td>numeric</td>
<td>.01</td>
<td></td>
</tr>
<tr>
<td>xvector</td>
<td>list</td>
<td>{ -0.7, -0.7 }</td>
<td></td>
</tr>
<tr>
<td>yvector</td>
<td>list</td>
<td>{ 1, 0 }</td>
<td></td>
</tr>
<tr>
<td>zvector</td>
<td>list</td>
<td>{ 0, 1 }</td>
<td></td>
</tr>
<tr>
<td>light</td>
<td>list</td>
<td>{ 3, 3, 10 }</td>
<td></td>
</tr>
<tr>
<td>bright</td>
<td>numeric</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>clip</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td>boolean</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>axis</td>
<td>list</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>clipaxis</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>axiscolor</td>
<td>string</td>
<td>&quot;gray&quot;</td>
<td></td>
</tr>
<tr>
<td>axislinewidth</td>
<td>numeric</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>
10 Mesh

This is more a gimmick than of real practical use. A mesh is a set of paths that gets transformed into hyperlinks. So, as a start you need to enable these:

\setupinteraction
[\setupinteraction{\state=start,\color=white,\contrastcolor=white}]

We just give a bunch of examples of meshes. A path is divided in smaller paths and each of them is part of the same hyperlink. An application is for instance clickable maps but (so far) only Acrobat supports such paths.

\startuseMPgraphic
{MyPath1}
\save p ; \path p[] ;
\p1 := \unitsquare \xysized( OverlayWidth/4, OverlayHeight/4 ) ;
\p2 := \unitsquare \xysized( OverlayWidth/4,3OverlayHeight/5 ) \shifted ( OverlayWidth/4,0 ) ;
\p3 := \unitsquare \xysized( OverlayWidth/4, OverlayHeight ) \shifted ( 3 OverlayWidth/4,0 ) ;
\fill p1 \withcolor "darkred" ;
\fill p2 \withcolor "darkblue" ;
\fill p3 \withcolor "darkgreen" ;
draw \lmt_mesh [ \paths = { p1, p2, p3 } ] ;
\setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

Such a definition is used as follows. First we define the mesh as overlay:

\defineoverlay[MyPath1][\useMPgraphic{MyPath1}]

Then, later on, this overlay can be used as background for a button. Here we just jump to another page. The rendering is shown in figure 10.1.

\button
[\height=3cm,\width=4cm,\background=MyPath1,\frame=off]
{Example 1}
[\realpage(2)]

More interesting are non-rectangular shapes so we show a bunch of them. You can pass multiple paths, influence the accuracy by setting the number of steps and show the mesh with the tracing option.

\startuseMPgraphic
{MyPath2}
\save q ; \path q ; q := \unitcircle \xysized(OverlayWidth,OverlayHeight) ;
\save p ; \path p ; p := \for i=1 \upto \length(q) :
Example 1

Figure 10.1

(\textbf{center} q) -- (\textbf{point} (i-1) of q) -- (\textbf{point} i of q) -- (\textbf{center} q) --
endfor cycle;
fill q \textbf{withcolor} "darkgray";
draw lmt\_mesh [\n  \textbf{trace} = \textbf{true},
  \textbf{paths} = \{ p \}\n] \textbf{withcolor} "darkred";
\setbounds \textbf{currentpicture} \textbf{to} OverlayBox;
\stopmpgraphic
\startmpgraphic{MyPath3}
 \save q ; \textbf{path} q ; q := \textbf{unitcircle xysized}(OverlayWidth,OverlayHeight) \textbf{randomized} 3mm ;
fill q \textbf{withcolor} "darkgray";
draw lmt\_mesh [\n  \textbf{trace} = \textbf{true},
  \textbf{paths} = \{ \textbf{meshed}(q,OverlayBox,.05) \}\n] \textbf{withcolor} "darkgreen";
% draw OverlayMesh(q,.025) withcolor "darkgreen";
\setbounds \textbf{currentpicture} \textbf{to} OverlayBox;
\stopmpgraphic
\startmpgraphic{MyPath4}
 \save q ; \textbf{path} q ; q := \textbf{unitcircle xysized}(OverlayWidth,OverlayHeight) \textbf{randomized} 3mm ;
fill q \textbf{withcolor} "darkgray";
draw lmt\_mesh [\n  \textbf{trace} = \textbf{true},
  \textbf{auto} = \textbf{true},
  \textbf{step} = 0.0125,
  \textbf{paths} = \{ q \}\n] \textbf{withcolor} "darkyellow";
\setbounds \textbf{currentpicture} \textbf{to} OverlayBox;
\stopmpgraphic
\startmpgraphic{MyPath5}
save q ; path q ; q := unitdiamond xysized(OverlayWidth,OverlayHeight) randomized 2mm ;
q := q shifted - center q shifted center OverlayBox ;
fill q withcolor "darkgray" ;
draw lmt_mesh [
  trace = true,
  auto = true,
  step = 0.0125,
  paths = { q }
] withcolor "darkmagenta" ;
setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic
\startuseMPgraphic{MyPath6}
save p ; path p[] ;
p1 := p2 := fullcircle xysized(20OverlayWidth/5,20OverlayHeight/3) ;
p1 := p1 shifted - center p1 shifted center OverlayBox shifted (-1 OverlayWidth/4,0) ;
p2 := p2 shifted - center p2 shifted center OverlayBox shifted ( 1 OverlayWidth/4,0) ;
fill p1 withcolor "middlegray" ;
fill p2 withcolor "middlegray" ;
draw lmt_mesh [
  trace = true,
  auto = true,
  step = 0.02,
  paths = { p1, p2 }
] withcolor "darkcyan" ;
setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic
\startuseMPgraphic{MyPath7}
save p ; path p[] ;
p1 := p2 := fullcircle xysized(20OverlayWidth/5,20OverlayHeight/3) rotated 45 ;
p1 := p1 shifted - center p1 shifted center OverlayBox shifted (-1 OverlayWidth/4,0) ;
p2 := p2 shifted - center p2 shifted center OverlayBox shifted ( 1 OverlayWidth/4,0) ;
fill p1 withcolor "middlegray" ;
fill p2 withcolor "middlegray" ;
draw lmt_mesh [
  trace = true,
  auto = true,
  step = 0.01,
  box = OverlayBox enlarged -5mm,
  paths = { p1, p2 }
] withcolor "darkcyan" ;
draw OverlayBox enlarged -5mm withcolor "darkgray" ;
This is typical a feature that, if used at all, needs some experimenting but at least the traced images look interesting enough. The six examples are shown in figure 10.2.

Figure 10.2
11 Function

It is tempting to make helpers that can do a lot. However, that also means that we need to explain a lot. Instead it makes more sense to have specific helpers and just make another one when needed. Rendering functions falls into this category. At some point users will come up with specific cases that other users can use. Therefore, the solution presented here is not the ultimate answer. We start with a simple example:

This image is defined as follows:

\begin{verbatim}
\startMPcode
{doublefun} 
    draw lmt_function [ 
        xmin = 0, xmax = 20, xstep = .1, 
        ymin = -2, ymax = 2, 
        sx = 1mm, xsmall = 80, xlarge = 20, 
        sy = 4mm, ysmall = 40, ylarge = 4, 
        linewidth = .025mm, offset = .1mm, 
        code = "1.5 * math.sind (50 * x - 150)" ,
    ] 
    xsized 8cm 
  ;
\stopMPcode
\end{verbatim}

We can draw multiple functions in one go. The next sample split the drawing over a few ranges and is defined as follows; in figure 11.2 we see the result.

\begin{verbatim}
\startMPcode
{doublefun} 
    draw lmt_function [ 
        xmin = 0, xmax = 20, xstep = .1, 
        ymin = -2, ymax = 2, 
        sx = 1mm, xsmall = 80, xlarge = 20, 
        sy = 4mm, ysmall = 40, ylarge = 4, 
        linewidth = .025mm, offset = .1mm, 
        code = "1.5 * math.sind (50 * x - 150)" ,
    ] 
    xsized 8cm 
  ;
\stopMPcode
\end{verbatim}
Instead of the same function, we can draw different ones and when we use transparency we get nice results too.

\definecolor[MyColorR][r=.5,t=.5,a=1]
\definecolor[MyColorG][g=.5,t=.5,a=1]
\definecolor[MyColorB][b=.5,t=.5,a=1]
\startMPcode{doublefun}
\draw lmt_function [ 
  xmin = 0, xmax = 20, xstep = .1, 
  ymin = -1, ymax = 1, 
  sx = 1mm, xsmall = 80, xlarge = 20, 
  sy = 4mm, ysmall = 40, ylarge = 4, 
  linewidth = .025mm, offset = .1mm, 
  functions = { 
    code = "\text{math.sind (50 \times x - 150)}", 
    close = true, 
    fillcolor = "MyColorR"
  },
\stopMPcode
It is important to choose a good step. In figure 11.4 we show 4 variants and it is clear that in this case using straight line segments is better (or at least more efficient with small steps).
You can manipulate the axis (a bit) by tweaking the first and last ticks. In the case of figure 11.5 we also put the shape on top of the axis.

\startMPcode{doublefun}
\draw lmt_function [  
  xfirst = 9, xlast = 21, ylarge = 2, ysmall = 1/5,  
  yfirst = -1, ylast = 1, xlarge = 2, xsmall = 1/4,  
  xmin = 10, xmax = 20, xstep = .25,  
  ymin = -1, ymax = 1,  
  drawcolor = "darkmagenta",  
  shape = "steps",  
  code = "0.5 * math.random(-2,2)",  
  linewidth = .025mm,  
  offset = .1mm,  
  reverse = true,  
]  
xsized TextWidth
;
\stopMPcode

The whole repertoire of parameters (in case of doubt just check the source code as this kind of code is not that hard to follow) is:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>linewidth</td>
<td></td>
<td>.025mm</td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td>.1mm</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
<td>&quot;math.sind (50 * x^2 - 150)&quot;</td>
<td></td>
</tr>
<tr>
<td>shape</td>
<td></td>
<td>&quot;curve&quot;</td>
<td></td>
</tr>
<tr>
<td>xsized</td>
<td></td>
<td>.45TextWidth</td>
<td></td>
</tr>
</tbody>
</table>
sx numeric 1mm horizontal scale factor
sy numeric 1mm vertical scale factor
offset numeric 0
xmin numeric 1
xmax numeric 1
xstep numeric 1
xsmall numeric optional step of small ticks
xlarge numeric optional step of large ticks
xlabels string no yes, no or nolimits
xticks string bottom possible locations are top, middle and bottom
Function

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Value/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcaption</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>ymin</td>
<td>numeric</td>
<td>1</td>
</tr>
<tr>
<td>ymax</td>
<td>numeric</td>
<td>1</td>
</tr>
<tr>
<td>ystep</td>
<td>numeric</td>
<td>1</td>
</tr>
<tr>
<td>ysmall</td>
<td>numeric</td>
<td>optional step of small ticks</td>
</tr>
<tr>
<td>ylarge</td>
<td>numeric</td>
<td>optional step of large ticks</td>
</tr>
<tr>
<td>xfirst</td>
<td>numeric</td>
<td>left of xmin</td>
</tr>
<tr>
<td>xlast</td>
<td>numeric</td>
<td>right of xmax</td>
</tr>
<tr>
<td>yfirst</td>
<td>numeric</td>
<td>below ymin</td>
</tr>
<tr>
<td>ylast</td>
<td>numeric</td>
<td>above ymax</td>
</tr>
<tr>
<td>ylabels</td>
<td>string</td>
<td>no, yes, or no limits</td>
</tr>
<tr>
<td>yticks</td>
<td>string</td>
<td>left, middle, and right</td>
</tr>
<tr>
<td>ycaption</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>shape</td>
<td>string</td>
<td>curve or line</td>
</tr>
<tr>
<td>fillcolor</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>drawsize</td>
<td>numeric</td>
<td>1</td>
</tr>
<tr>
<td>drawcolor</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>frame</td>
<td>string</td>
<td>options are yes, ticks and sticks</td>
</tr>
<tr>
<td>linewidth</td>
<td>numeric</td>
<td>.05mm</td>
</tr>
<tr>
<td>pointsymbol</td>
<td>string</td>
<td>like type dots</td>
</tr>
<tr>
<td>pointsize</td>
<td>numeric</td>
<td>2</td>
</tr>
<tr>
<td>pointcolor</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>xarrow</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>yarrow</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>boolean</td>
<td>false</td>
</tr>
</tbody>
</table>

When true, draw the function between axis and labels.
12 Chart

This is another example implementation but it might be handy for simple cases of presenting results. Of course one can debate the usefulness of certain ways of presenting but here we avoid that discussion. Let’s start with a simple pie chart (figure 12.1).

\startMPcode
draw lmt_chart_circle [    samples = { { 1, 4, 3, 2, 5, 7, 6 } },    percentage = true,    trace = true,
]; \stopMPcode

Figure 12.1

As with all these lmtx extensions, you’re invited to play with the parameters. in figure 12.2 we see a variant that adds labels as well as one that has a legend.

The styling of labels and legends can be influenced independently.

\startMPcode
draw lmt_chart_circle [    height = 4cm,    samples = { { 1, 4, 3, 2, 5, 7, 6 } },    percentage = true,    trace = true,    labelcolor = "white",    labelformat = "@0.1f",    labelstyle = "tttx"
]; \stopMPcode

\startMPcode
draw lmt_chart_circle [    height = 4cm,    samples = { { 1, 4, 3, 2, 5, 7, 6 } },    percentage = false,    trace = true,
]; \stopMPcode
A second way of rendering are histograms, and the interface is mostly the same. In figure 12.3 we see two variants.

```
\startMPcode
  draw lmt_chart_histogram [
    samples     = { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage  = true,
    cumulative  = true,
    trace       = true,
  ] ;
\stopMPcode
```

and one with two datasets:

```
\startMPcode
  draw lmt_chart_histogram [
    samples     = {
      { 1, 4, 3, 2, 5, 7, 6 },
      { 1, 2, 3, 4, 5, 6, 7 }
    },
    background  = "lightgray",
    trace       = true,
  ] ;
\stopMPcode
```
A cumulative variant is shown in figure 12.4 where we also add a background (color).

\startMPpage[offset=5mm]
   draw lmt_chart_histogram [  
      samples = {
         { 1, 4, 3, 2, 5, 7, 6 },  
         { 1, 2, 3, 4, 5, 6, 7 }  
      },  
      percentage = true,  
      cumulative = true,  
      showlabels = false,  
      backgroundcolor = "lightgray",  
   ];
\stopMPpage

A different way of using colors is shown in figure 12.5 where each sample gets its own (same) color.

\startMPcode
   draw lmt_chart_histogram [  
      samples = {
         { 1, 4, 3, 2, 5, 7, 6 },  
         { 1, 2, 3, 4, 5, 6, 7 }  
      }  
   ];
\stopMPcode
As with pie charts you can add labels and a legend:

\startMPcode
draw lmt_chart_histogram [
  height = 6cm,
  samples = { { 1, 4, 3, 2, 5, 7, 6 } },
  percentage = true,
  cumulative = true,
  trace = true,
  labelstyle = "ttxx",
  labelanchor = "top",
  labelcolor = "white",
  backgroundcolor = "middlegray",
] ;
\stopMPcode

The previous and next examples are shown in figure 12.6. The height specified here concerns the graphic and excludes the labels,

\startMPcode
draw lmt_chart_histogram [
  height = 6cm,
  width = 10mm,
  samples = { { 1, 4, 3, 2, 5, 7, 6 } },
  trace = true,
  maximum = 7.5,
  linewidth = 1mm,
  originsize = 0,
  labelanchor = "bot",
] ;
\stopMPcode
The third category concerns bar charts that run horizontal. Again we see similar options driving the rendering (figure 12.7).

Determining the offset of labels is manual work:
Here is one with a legend (rendered in figure 12.8):

\startMPcode
draw lmt_chart_bar [ \\
  width = 8cm, \\
  height = 10mm, \\
  samples = { { 1, 4, 3, 2, 5, 7, 6 } }, \\
  trace = true, \\
  maximum = 7.5, \\
  linewidth = 1mm, \\
  originsize = 0, \\
  labelanchor = "lft", \\
  labelcolor = "black" \\
  labelstyle = "bfxx" \\
  legendstyle = "tfxx", \\
  labelstrut = "yes", \\
  legend = { "first", "second", "third", "fourth", "fifth", "sixths", "sevenths" \\
  } \\
] ; 
\stopMPcode
You can have labels per dataset as well as draw multiple datasets in one image, see figure 12.9:

\startMPcode
\draw lmt_chart_bar [ samples = {
  \{ 1, 4, 3, 2, 5, 7, 6 \},
  \{ 3, 2, 5, 7, 5, 6, 1 \}\},
labels = {
  \{ "a1", "b1", "c1", "d1", "e1", "f1", "g1" \},
  \{ "a2", "b2", "c2", "d2", "e2", "f2", "g2" \}\},
labeloffset = -EmWidth, 
labelanchor = "center",
labelstyle = "ttxx", 
trace = true, 
center = true, ] ; 
\draw lmt_chart_bar [ samples = { 
  \{ 1, 4, 3, 2, 5, 7, 6 \}\},
labels = { 
  \{ "a", "b", "c", "d", "e", "f", "g" \}\},
labeloffset = -EmWidth, 
labelanchor = "center", 
trace = true, 
center = true, ] shifted (10cm,0) ; 
\stopMPcode
Figure 12.9

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>originsize</td>
<td>numeric</td>
<td>1mm</td>
<td></td>
</tr>
<tr>
<td>trace</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>showlabels</td>
<td>boolean</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>center</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>samples</td>
<td>list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>percentage</td>
<td>boolean</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>maximum</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>distance</td>
<td>numeric</td>
<td>1mm</td>
<td></td>
</tr>
<tr>
<td>labels</td>
<td>list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labelstyle</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labelformat</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labelstrut</td>
<td>string</td>
<td>auto</td>
<td></td>
</tr>
<tr>
<td>labelanchor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labeloffset</td>
<td>numeric</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>labelfraction</td>
<td>numeric</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>labelcolor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>backgroundcolor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>drawcolor</td>
<td>string</td>
<td>white</td>
<td></td>
</tr>
<tr>
<td>fillcolors</td>
<td>list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>colormode</td>
<td>string</td>
<td>global</td>
<td>or local</td>
</tr>
<tr>
<td>linewidth</td>
<td>numeric</td>
<td>.25mm</td>
<td></td>
</tr>
<tr>
<td>legendcolor</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>legendstyle</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>legend</td>
<td>list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pie charts have:

<table>
<thead>
<tr>
<th>name</th>
<th>default</th>
</tr>
</thead>
</table>
Histograms come with:

<table>
<thead>
<tr>
<th>name</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>5cm</td>
</tr>
<tr>
<td>width</td>
<td>5mm</td>
</tr>
<tr>
<td>labelanchor</td>
<td>bot</td>
</tr>
<tr>
<td>labeloffset</td>
<td>1mm</td>
</tr>
<tr>
<td>labelstrut</td>
<td>auto</td>
</tr>
</tbody>
</table>

Bar charts use:

<table>
<thead>
<tr>
<th>name</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>5cm</td>
</tr>
<tr>
<td>width</td>
<td>5mm</td>
</tr>
<tr>
<td>labelanchor</td>
<td>lft</td>
</tr>
<tr>
<td>labeloffset</td>
<td>1mm</td>
</tr>
<tr>
<td>labelstrut</td>
<td>no</td>
</tr>
</tbody>
</table>
13 SVG

There is not that much to tell about this command. It translates an svg image to MetaPost operators. We took a few images from a mozilla emoji font:

\startMPcode
\begin{verbatim}
draw lmt_svg [
  filename = "mozilla-svg-002.svg",
  height = 2cm,
  width = 8cm,
];
\end{verbatim}
\stopMPcode

Because we get pictures, you can mess around with them:

\startMPcode
\begin{verbatim}
picture p ; p := lmt_svg [ filename = "mozilla-svg-001.svg" ] ;
numeric w ; w := bbwidth(p) ;
draw p ;
draw p xscaled -1 shifted (2.5*w,0);
draw p rotatedaround(center p,45) shifted (3.0*w,0) ;
draw image (for i within p : if filled i :
  draw pathpart i withcolor green ;
fi endfor ;
) shifted (4.5*w,0);
draw image (for i within p : if filled i :
  fill pathpart i withcolor red withtransparency (1,.25) ;
fi endfor ;
) shifted (6*w,0);
\end{verbatim}
\stopMPcode

Of course, often you won't know in advance what is inside the image and how (well) it has been defined so the previous example is more about showing some MetaPost muscle.

The supported parameters are:
<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>numeric</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
14 Fonts

Fonts are interesting phenomena but can also be quite hairy. Shapes can be missing or not to your liking. There can be bugs too. Control over fonts has always been on the agenda of \TeX{} macro packages, and ConTeXt provides a lot of control, especially in MkIV. In lmtx we add some more to that: we bring back \textsc{MetaFont}’s but now in the MetaPost way. A simple example shows how this is (maybe I should say: will be) done.

We define three simple shapes and do that (for now) in the \texttt{simplefun} instance because that’s what is used when generating the glyphs.

\verbatim
\startMPcalculation{simplefun}
  vardef TestGlyphLB =
    image (
      fill (unitsquare xscaled 10 yscaled 16 shifted (0,-3))
        withcolor "darkred" withtransparency (1,.5)
    );
  enddef;

  vardef TestGlyphRB =
    image (
      fill (unitcircle xscaled 15 yscaled 12 shifted (0,-2))
        withcolor "darkblue" withtransparency (1,.5)
    );
  enddef;

  vardef TestGlyphFS =
    image (
      fill (unittriangle xscaled 15 yscaled 12 shifted (0,-2))
        withcolor "darkgreen" withtransparency (1,.5)
    );
  enddef;
\stopMPcalculation

This is not that spectacular, not is the following:

\verbatim
\startMPcalculation{simplefun}
  lmt_registerglyphs [
    name = "test",
    units = 10, % 1000
  ];

  lmt_registerglyph [
    category = "test",
    unicode = 123,
    code = "draw TestGlyphLB ;",
\stopMPcalculation
width = 10, % 1000
height = 13, % 1300
depth = 3 % 300
]

lmt_registerglyph [  
category = "test",  
unicode = 125,  
code = "draw TestGlyphRB ;",  
width = 15,  
height = 10,  
depth = 2
]

lmt_registerglyph [  
category = "test",  
unicode = "/",  
code = "draw TestGlyphFS ;",  
width = 15,  
height = 10,  
depth = 2
]

\stopMPcalculation

We now define a font. We always use a font as starting point which has the advantage that we always get something reasonable when we test. Of course you can use this mps font feature with other fonts too.

\definefontfeature[metapost][metapost=test] % or: mps={category=test}

\definefont[MyFontA][Serif*metapost @ 10bp]
\definefont[MyFontB][Serif*metapost @ 12bp]

These fonts can now be used:

\MyFontA \dorecurse{20}{\{}#/1/\}\}\par
\MyFontB \dorecurse{20}{\{}#/1/\}\}\par

We get some useless text but it demonstrates the idea:
If you know a bit more about ConTeXt you could think: so what, wasn’t this already possible? Sure, there are various ways to achieve similar effects, but the method described here has a few advantages: it’s relatively easy and we’re talking about real fonts here. This means that using the shapes for characters is pretty efficient.

A more realistic example is given next. It is a subset of what is available in the ConTeXt core.

\startMPcalculation{simplefun}

pen SymbolPen ; SymbolPen := pencircle scaled 1/4 ;

vardef SymbolBullet =
    fill unitcircle scaled 3 shifted (1.5,1.5) withpen SymbolPen
enddef ;
vardef SymbolSquare =
    draw unitssquare scaled (3-1/16) shifted (1.5,1.5) withpen SymbolPen
enddef ;
vardef SymbolBlackDiamond =
    fillup unitdiamond scaled (3-1/16) shifted (1.5,1.5) withpen SymbolPen
enddef ;
vardef SymbolNotDef =
    draw center unitcircle
    scaled 3 shifted (1.5,1.5)
    withpen SymbolPen scaled 4
enddef ;
lmt_registerglyphs [   name = "symbols",
    units = 10,
    usecolor = true,
    width = 6,
    height = 6,
    depth = 0,
    code = "SymbolNotDef ;",
] ;
lmt_registerglyph [  category = "symbols", unicode = "0x2022",
    code = "SymbolBullet ;"
] ;
lmt_registerglyph [  category = "symbols", unicode = "0x25A1",
    code = "SymbolSquare ;"
] ;
lmt_registerglyph [  category = "symbols", unicode = "0x25C6",
    code = "SymbolBlackDiamond ;"
] ;
\stopMPcalculation

We could use these symbols in for instance itemize symbols. You might notice the potential difference in bullets:
\definefontfeature{metapost}[metapost=symbols]
\definefont{MyFont}[Serif*metapost sa 1]

\startitemize[packed]
\startitem{\MyFont} • \quad \text{Regular rendering.}\stopitem
\startitem{\MyFont\red} • \quad \text{Rendering with color.}\stopitem
\startitem{\MyFont\blue\showglyphs} • \quad \text{Idem but with boundingboxes shown.}\stopitem
\stopitemize

\begin{itemize}
\item Regular rendering.
\item Rendering with color.
\item Idem but with boundingboxes shown.
\end{itemize}

When blown up, these symbols look as follows:

\begin{figure}
\centering
\begin{tabular}{c|c|c}
\hline
\hline
\hline
\end{tabular}
\end{figure}

You can use these tricks with basically any font, so also with math fonts. However, at least for now, you need to define these before the font gets loaded.

\startMPcalculation\{simplefun\}
\textsf{pen KindergartenPen ; KindergartenPen := pencircle scaled 1 ;}
\textsf{\% 10 x 10 grid}
\textsf{vardef KindergartenEqual =}
\textsf{draw image}
\textsf{\quad (draw (2,6) -- (9,5) ;}
\textsf{\quad (draw (2,4) -- (8,3) ;}
\textsf{\quad shifted (0,-2) withpen KindergartenPen withcolor "KindergartenEqual" enddef ;}
\textsf{vardef KindergartenPlus =}
\textsf{draw image}
\textsf{\quad (draw (1,4) -- (9,5) ;}
\textsf{\quad (draw (4,1) -- (5,8) ;}
\textsf{\quad shifted (0,-2) withpen KindergartenPen withcolor "KindergartenPlus" enddef ;}
vardef KindergartenMinus =
  draw image
  (
    draw (1,5) -- (9,4);
  )
  shifted (0,-2)
  withpen KindergartenPen
  withcolor "KindergartenMinus"
enddef;
vardef KindergartenTimes =
  draw image
  (
    draw (2,1) -- (9,8);
    draw (8,1) -- (2,8);
  )
  shifted (0,-2)
  withpen KindergartenPen
  withcolor "KindergartenTimes"
enddef;
vardef KindergartenDivided =
  draw image
  (
    draw (2,1) -- (8,9);
  )
  shifted (0,-2)
  withpen KindergartenPen
  withcolor "KindergartenDivided"
enddef;
lmt_registerglyphs [  
  name   = "kindergarten",
  units  = 10,
  % usecolor = true,
  width  = 10,
  height = 8,
  depth  = 2,
  ];
lmt_registerglyph [ category = "kindergarten", unicode = "0x003D",
  code = "KindergartenEqual" ];
lmt_registerglyph [ category = "kindergarten", unicode = "0x002B",
  code = "KindergartenPlus" ];
lmt_registerglyph [ category = "kindergarten", unicode = "0x2212",
  code = "KindergartenMinus" ];
lmt_registerglyph [ category = "kindergarten", unicode = "0x00D7",
  code = "KindergartenTimes" ];

Fonts uncorrected draft 63
We also define the colors. If we leave usecolor to true, the text colors will be taken.

\definecolor\[KindergartenEqual\] [darkgreen]
\definecolor\[KindergartenPlus\] [darkred]
\definecolor\[KindergartenMinus\] [darkred]
\definecolor\[KindergartenTimes\] [darkblue]
\definecolor\[KindergartenDivided\] [darkblue]

\definefontfeature\[mathextra\][metapost=kindergarten]

Here is an example:

\switchtobodyfont[cambria]

$ y = 2 \times x + a - b / 3 $

Scaled up:

\[
\begin{align*}
  \textcolor{darkgreen}{y} & = \textcolor{darkred}{2} \times \textcolor{black}{x} + \textcolor{darkblue}{a} - \textcolor{darkred}{b} / 3 \\
\end{align*}
\]

Of course this won’t work out well (yet) with extensible yet, due to related definitions for which we don’t have an interface yet. There is one thing that you need to keep in mind: the fonts are flushed when the document gets finalized so you have to make sure that colors are defined at the level that they are still valid at that time. So best put color definitions like the above in the document style.

This is an experimental interface anyway so we don’t explain the parameters yet as there might be more of them.
15 Interface

Because graphic solutions are always kind of personal or domain driven it makes not much sense to cook up very generic solutions. If you have a project where MetaPost can be of help, it also makes sense to spend some time on implementing the basics that you need. In that case you can just copy and tweak what is there. The easiest way to do that is to make a test file and use:

\startMPpage
  \% your code
\stopMPpage

Often you don’t need to write macros, and standard drawing commands will do the job, but when you find yourself repeating code, a wrapper might make sense. And this is why we have this key/value interface: it’s easier to abstract your settings than to pass them as (expression or text) arguments to a macro, especially when there are many.

You can find many examples of the key/value driven user interface in the source files and these are actually not that hard to understand when you know a bit of MetaPost and the additional macros that come with MetaFun. In case you wonder about overhead: the performance of this mechanism is pretty good.

Although the parameter handler runs on top of the Lua interface, you don’t need to use Lua unless you find that MetaPost can’t do the job. I won’t give examples of coding because I think that the source of MetaFun provides enough clues, especially the file mp-lmtx.mpxl. As the name suggests this is part of the ConTExt version lmtx, which runs on top of LuaMetaTEx. I leave it open if I will backport this functionality to LuaTEx and therefore MkIV.

An excellent explanation of this interface can be found at:

https://adityam.github.io/context-blog/post/new-metafun-interface/

So (at least for now) here I can stick to just mentioning the currently stable interface macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>presetparameters</td>
<td>Assign default values to a category of parameters. Sometimes it makes sense</td>
</tr>
<tr>
<td></td>
<td>not to set a default, because then you can check if a parameter has been</td>
</tr>
<tr>
<td></td>
<td>set at all.</td>
</tr>
<tr>
<td>applyparameters</td>
<td>This prepares the parameter handler for the given category and calls the</td>
</tr>
<tr>
<td></td>
<td>given macro when that is done.</td>
</tr>
<tr>
<td>getparameters</td>
<td>The parameters given after the category name are set.</td>
</tr>
<tr>
<td>hasparameter</td>
<td>Returns true when a parameter is set, and false otherwise.</td>
</tr>
<tr>
<td>hasoption</td>
<td>Returns true when there is overlap in given options, and false otherwise.</td>
</tr>
<tr>
<td>getparameter</td>
<td>Resolves the parameter with the given name.</td>
</tr>
</tbody>
</table>

because a parameter itself can have a parame-
<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getparameterdefault</td>
<td>names</td>
<td>Resolves the parameter with the given name. Because a parameter itself can have a parameter list you can pass additional names to reach the final destination. The last value is used when no parameter is found.</td>
</tr>
<tr>
<td>getparametercount</td>
<td>names</td>
<td>Returns the size if a list (array).</td>
</tr>
<tr>
<td>getmaxparametercount</td>
<td>names</td>
<td>Returns the size if a list (array) but descends into lists to find the largest size of a sublist.</td>
</tr>
<tr>
<td>getparameterpath</td>
<td>names, string, boolean</td>
<td>Returns the parameter as path. The optional string is one of -- or .. and the also optional boolean will force a closed path.</td>
</tr>
<tr>
<td>getparameterpen</td>
<td>names</td>
<td>Returns the parameter as pen (path).</td>
</tr>
<tr>
<td>getparameterext</td>
<td>names, boolean</td>
<td>Returns the parameter as string. The boolean can be used to force prepending a so called \strut.</td>
</tr>
<tr>
<td>pushparameters</td>
<td>category</td>
<td>Pushed the given (sub) category onto the stack so that we don't need to give the category each time.</td>
</tr>
<tr>
<td>popparameters</td>
<td></td>
<td>Pops the current (sub) category from the stack.</td>
</tr>
</tbody>
</table>

Most commands accept a list of strings separated by one or more spaces, The resolved will then step-wise descend into the parameter tree. This means that a parameter itself can refer to a list. When a value is an array and the last name is a number, the value at the given index will be returned.

> "category" "name" ... "name"
> "category" "name" ... number

The category is not used when we have pushed a (sub) category which can save you some typing and also is more efficient. Of course than can mean that you need to store values at a higher level when you need them at a deeper level.

There are quite some extra helpers that relate to this mechanism, at the MetaPost end as well as at the Lua end. They aim for instance at efficiently dealing with paths and can be seen at work in the mentioned module.

There is one thing you should notice. While MetaPost has numeric, string, boolean and path variables that can be conveniently be passed to and from Lua, communicating colors is a bit of a hassle. This is because rgb and cmyk colors and gray scales use different types. For this reason it is strongly recommended to use strings that refer to predefined colors instead. This also enforces consistency with the \TeX end. As convenience you can define colors at the MetaFun end.

\startMPcode
    definecolor [ name = "MyColor", r = .5, g = .25, b = .25 ]

    fill fullsquare xyscaled (TextWidth,5mm) withcolor "MyColor" ;
\stopMPcode