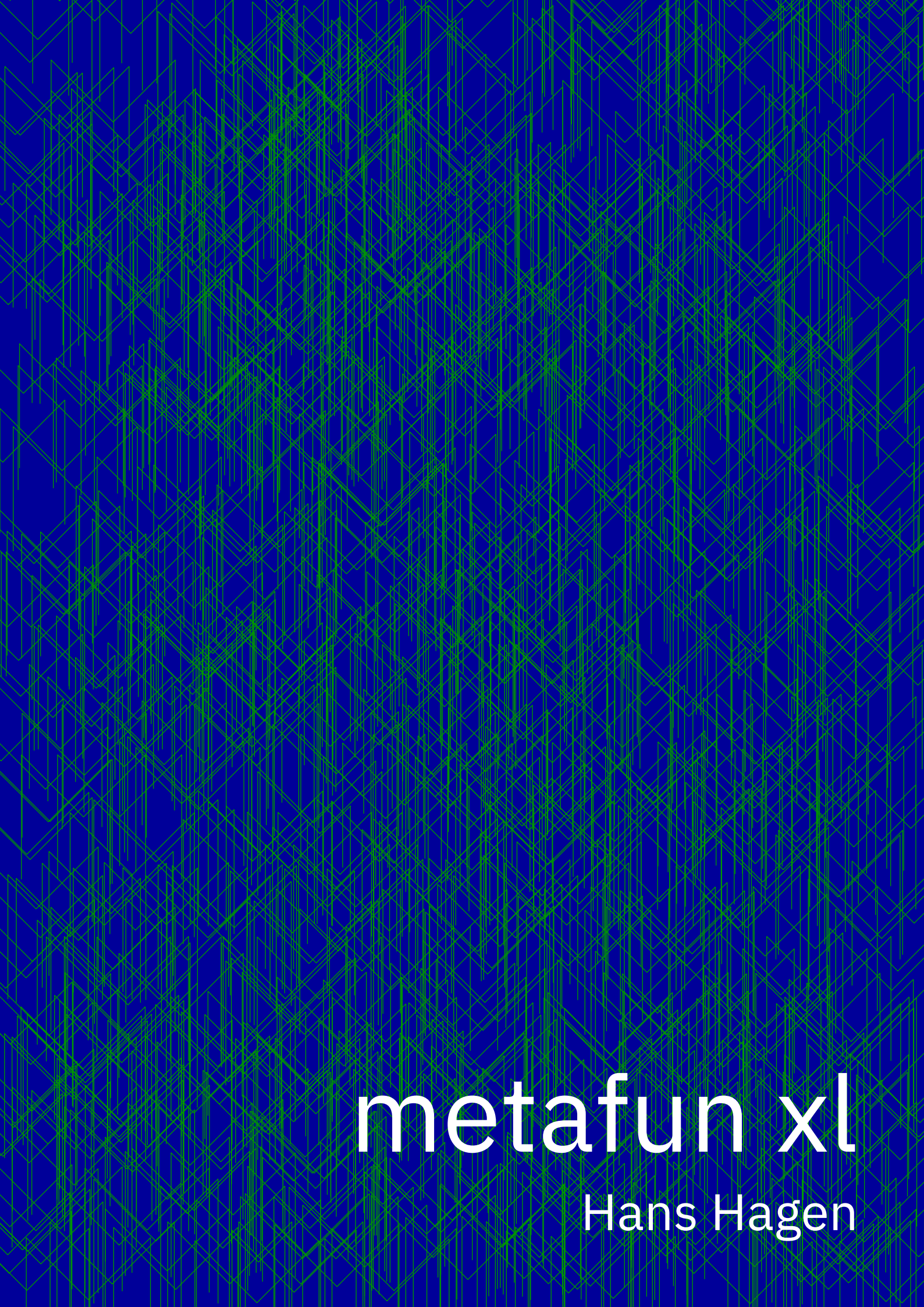

metafun xl
Hans Hagen

Contents

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 lmtx 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:

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

```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw textext("\framed{\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.

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:

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",
```

```
        style  = "bold",
        format = "@0.3F",
    ] ;
\stopMPcode
```

The following parameters can be set:

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

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 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:

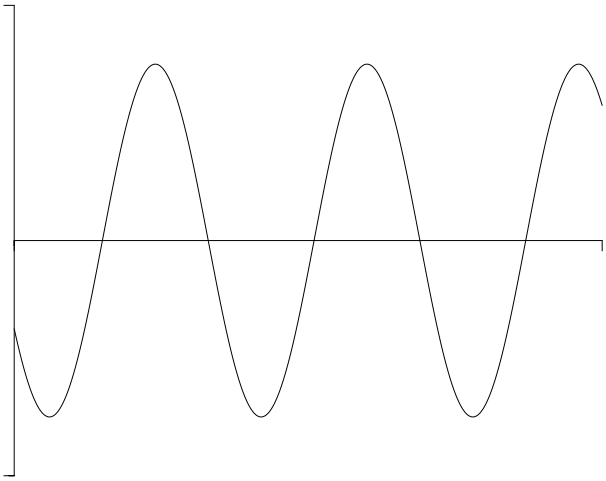


Figure 2.1

This image is defined as follows:

```
\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
```

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 2.2 we see the result.

```
\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,

        xticks    = "bottom",
        yticks    = "left",
        xlabels   = "nolimits",
        ylabels   = "yes",
        code      = "1.5 * math.sind (50 * x - 150)",
      % frame     = "ticks",
        frame     = "sticks",
        ycaption  = "\strut \rotate[rotation=90]{something vertical, using
          $\sin{x}$}",
        xcaption  = "\strut something horizontal",
        functions = {
            [ xmin =  1.0, xmax =  7.0, close = true, fillcolor = "darkred" ],
            [ xmin =  7.0, xmax = 12.0, close = true, fillcolor = "darkgreen" ],
            [ xmin = 12.0, xmax = 19.0, close = true, fillcolor = "darkblue" ],
            [
                drawcolor = "darkyellow",
                drawsize  = 2
            ]
        }
    ]
        xsized TextWidth
    ;
\stopMPcode
```

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      = "math.sind (50 * x - 150)",
                close     = true,
                fillcolor = "MyColorR"
            ],
```

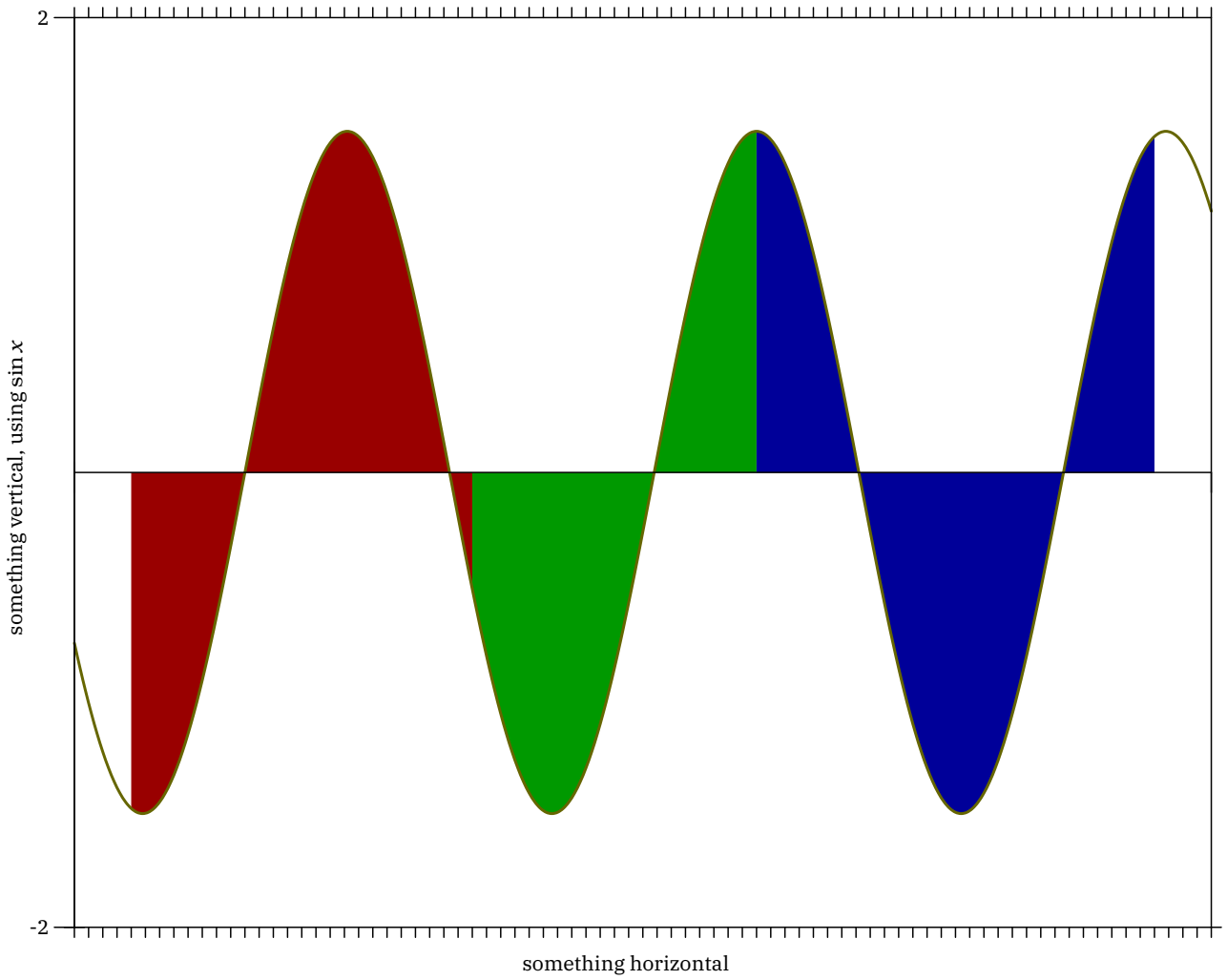


Figure 2.2

```
            [
                code      = "math.cosd (50 * x - 150)",
                close     = true,
                fillcolor = "MyColorB"
            ]
        },
    ]
        xsized TextWidth
    ;
\stopMPcode
```

It is important to choose a good step. In figure 2.4 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).

```
\startMPcode{doublefun}
    draw lmt_function [
        xmin =  0, xmax = 10, xstep = .1,
        ymin = -1, ymax =  1,

        sx = 1mm, sy = 4mm,
```

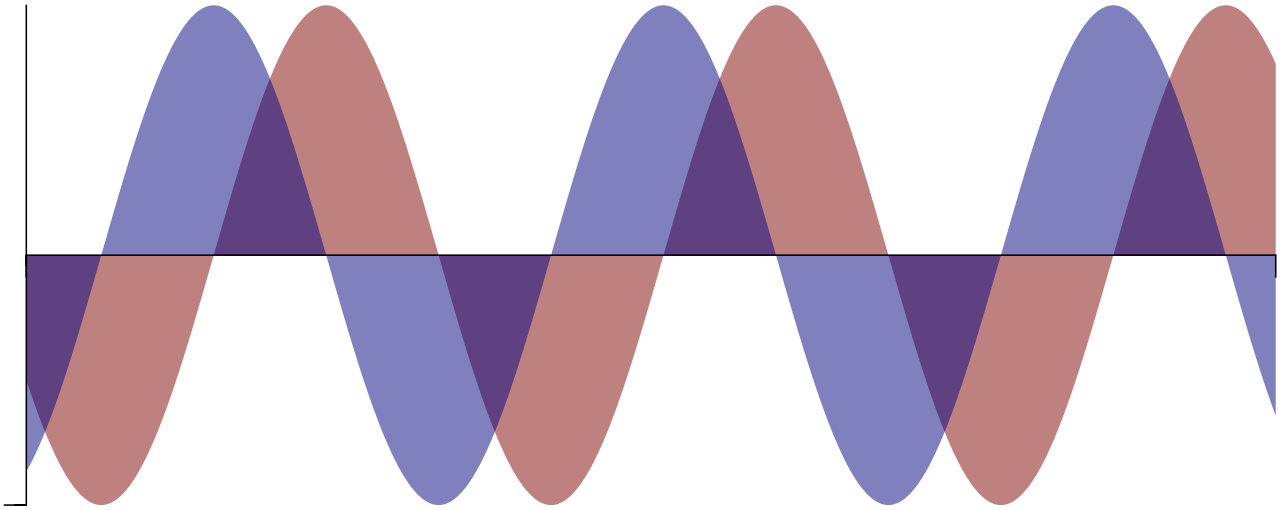


Figure 2.3

```
        linewidth = .025mm, offset = .1mm,

        code  = "math.sind (50 * x^2 - 150)",
        shape = "curve"
    ]
        xsized .45TextWidth
    ;
\stopMPcode
```

You can manipulate the axis (a bit) by tweaking the first and last ticks. In the case of figure 2.5 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:

name	type	default	comment

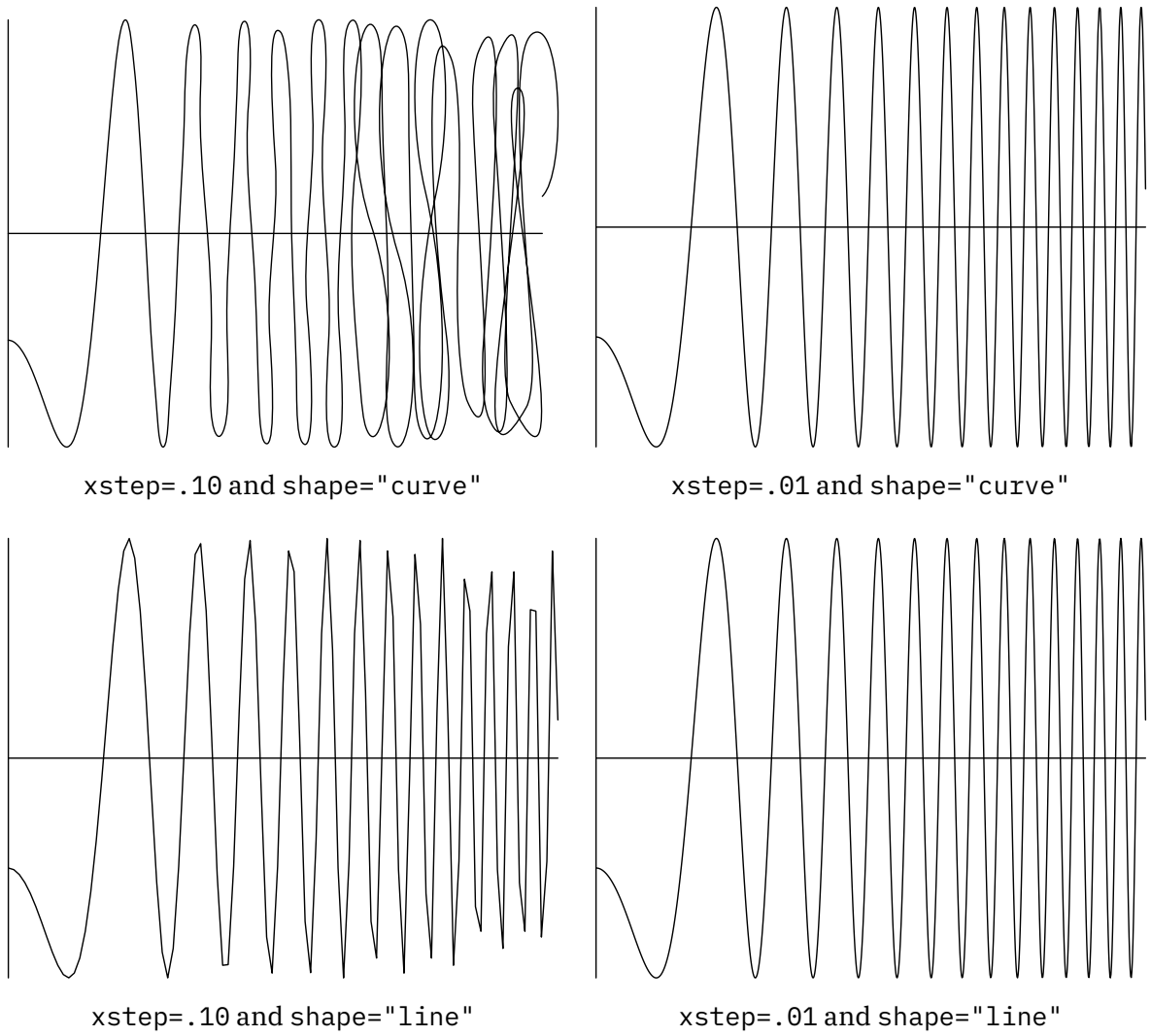


Figure 2.4

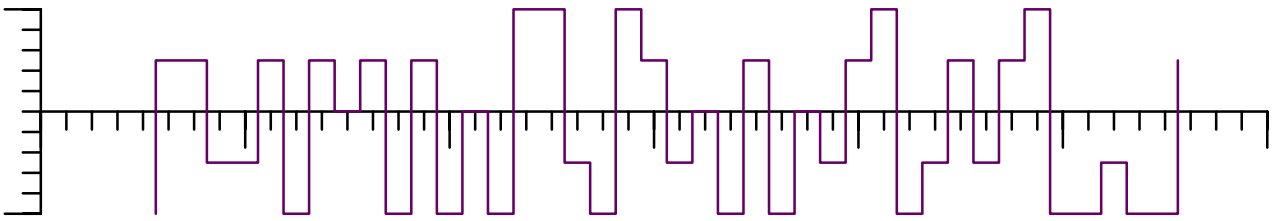


Figure 2.5

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

xcaption	string		
ymin	numeric	1	
ymax	numeric	1	
ystep	numeric	1	
ysmall	numeric		optional step of small ticks
ylarge	numeric		optional step of large ticks
xfirst	numeric		left of xmin
xlast	numeric		right of xmax
yfirst	numeric		below ymin
ylast	numeric		above ymax
ylabels	string	no	yes, no or nolimits
yticks	string	left	possible locations are left, middle and right
ycaption	string		
code	string		
close	boolean	false	
shape	string	curve	or line
fillcolor	string		
drawsize	numeric	1	
drawcolor	string		
frame	string		options are yes, ticks and sticks
linewidth	numeric	.05mm	
pointsymbol	string		like type dots
pointsize	numeric	2	
pointcolor	string		
xarrow	string		
yarrow	string		
reverse	boolean	false	when true draw the function between axis and labels

3 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 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.

```
\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",
```

[1] I have to figure out how to improve it a bit so that multiple path don't get connected.

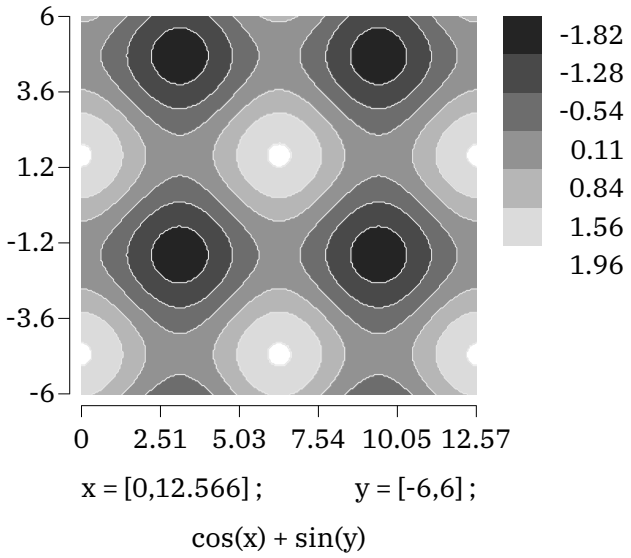


Figure 3.1

```
        foreground = "edge",
        linewidth  = 1/2,
        cache      = true,
    ] ;
\stopMPcode
```

In figure 3.1 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 3.2 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 3.3. 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.

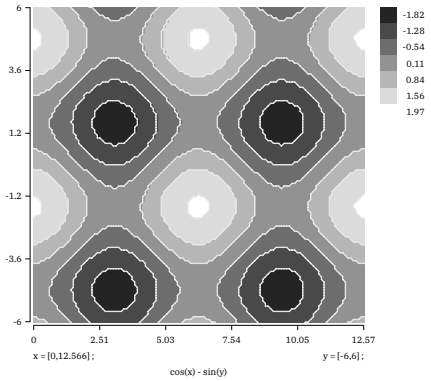


Figure 3.2

```
\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
```

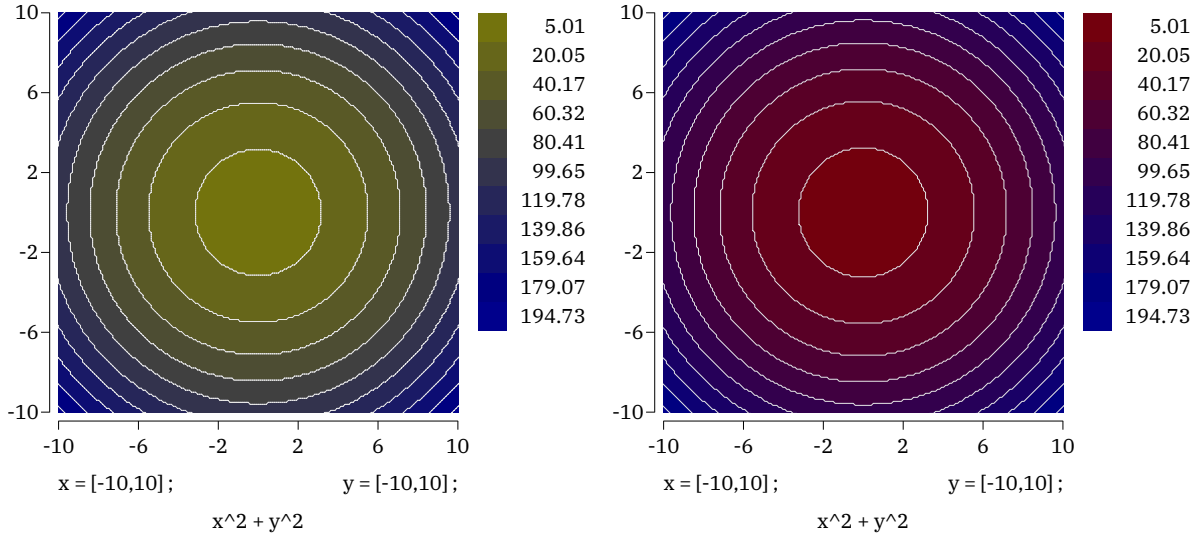


Figure 3.3

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:

```
\startluacode
    function document.MyContourA(x,y)
        return x^2 + y^2
    end
\stopluacode
```

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 `lin` to calculate a fraction of the current level and total number of levels.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -3, xmax = 3, xstep =  .01,
        ymin = -1, ymax = 1, ystep =  .01,

        levels     = 10,
        default    = .5,
        height     = 5cm,
        function   = "x^2 + y^2 + x + y/2",
        color      = "lin(l), 0, 1/2",
        background = "bitmap"
        foreground = "none",
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

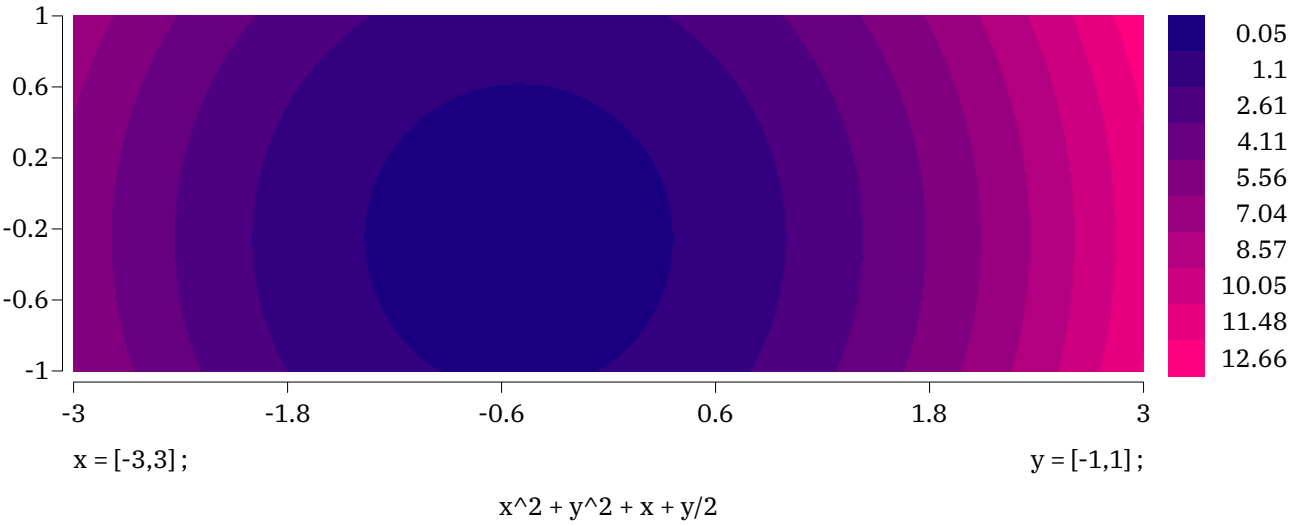


Figure 3.4

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 3.5.

```
\startMPcode{doublefun}
    draw lmt_contour [
```

```
        xmin = -3, xmax = 3, xstep =  .01,
        ymin = -1, ymax = 1, ystep =  .01,

        levels     = 10,
        default    = .5,
        height     = 5cm,
        function   = "x^2 + y^2 + x + y/2",
        color      = "lin(l), 1/2, 0",
        background = "band",
        foreground = "none",
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

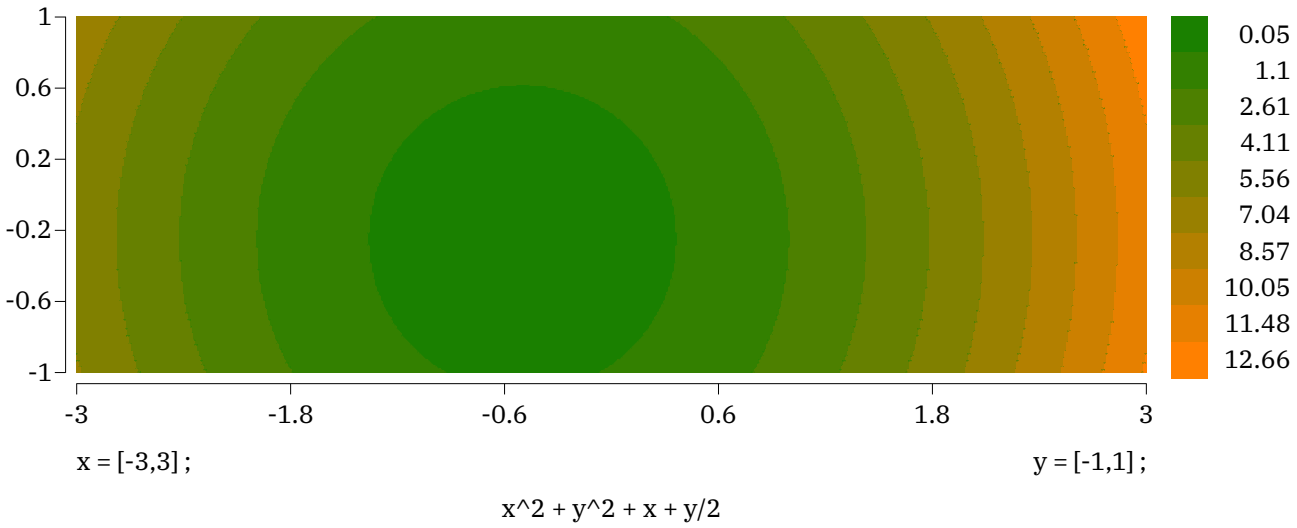


Figure 3.5

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      = { -.1, .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})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

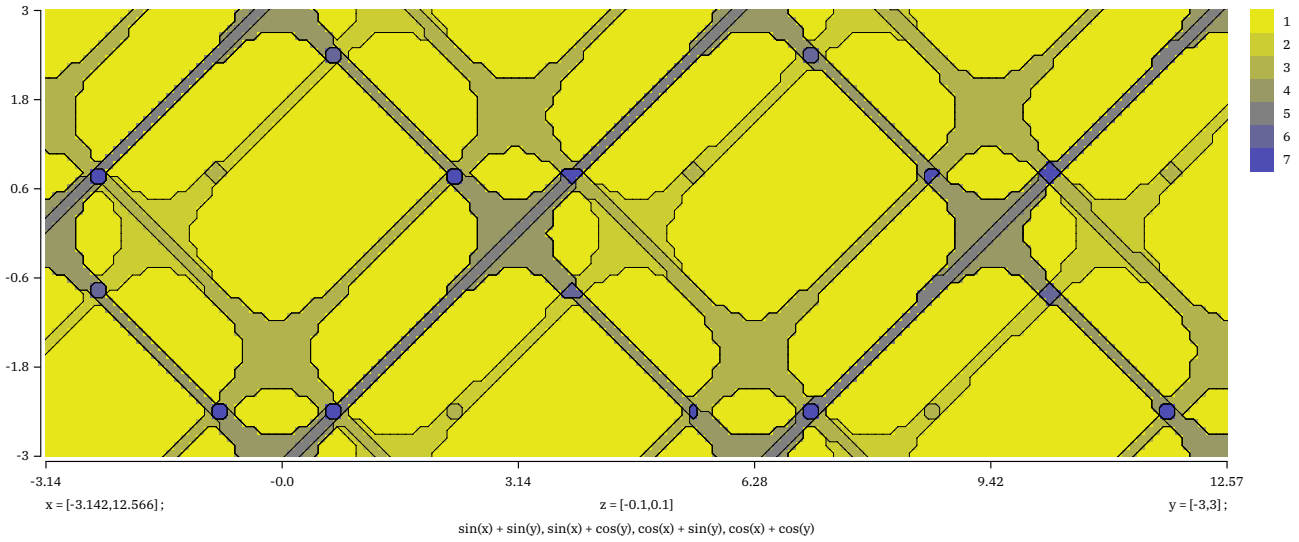


Figure 3.6

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 3.7 this is demonstrated. There we also show a variant foreground `cell`, which uses a bit different method for calculating the edges.[2]

```
\startMPcode{doublefun}
    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 ;
\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.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -2*pi, xmax =  2*pi, xstep = .01,
        ymin = -3,    ymax =  3,    ystep = .01,
```

[2] This a bit of a playground: more variants might show up in due time.

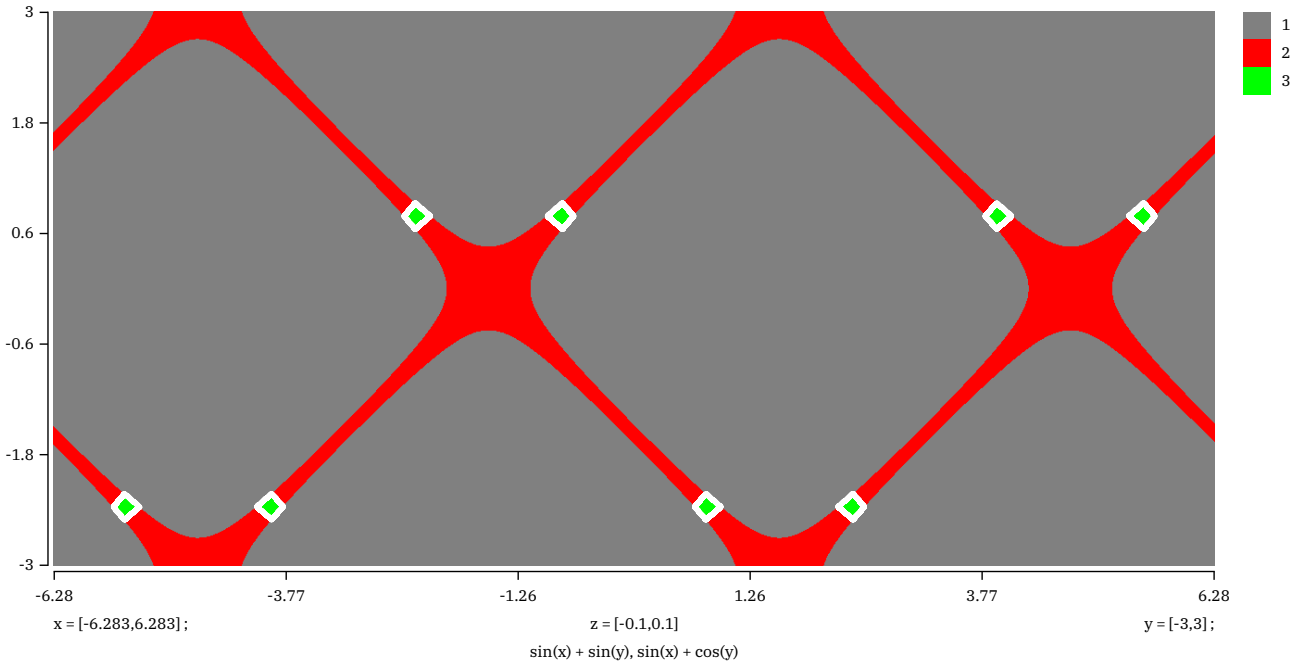


Figure 3.7

```
        range      = { -.1, .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",
        foreground = "none",
        level      = 3,
        color      = "shade({2/3,0,0},{2/3,1,2/3})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

Of course one can wonder how useful showing many functions but it can give nice pictures, as shown in figure 3.8.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin = -2*pi, xmax =  2*pi, xstep = .01,
        ymin = -3,    ymax =  3,    ystep = .01,

        range      = { -.3, .3 },
        preamble   = "local sin, cos = math.sin, math.cos",
        functions  = {
            "sin(x) + sin(y)",
            "sin(x) + cos(y)",
            "cos(x) + sin(y)",
```

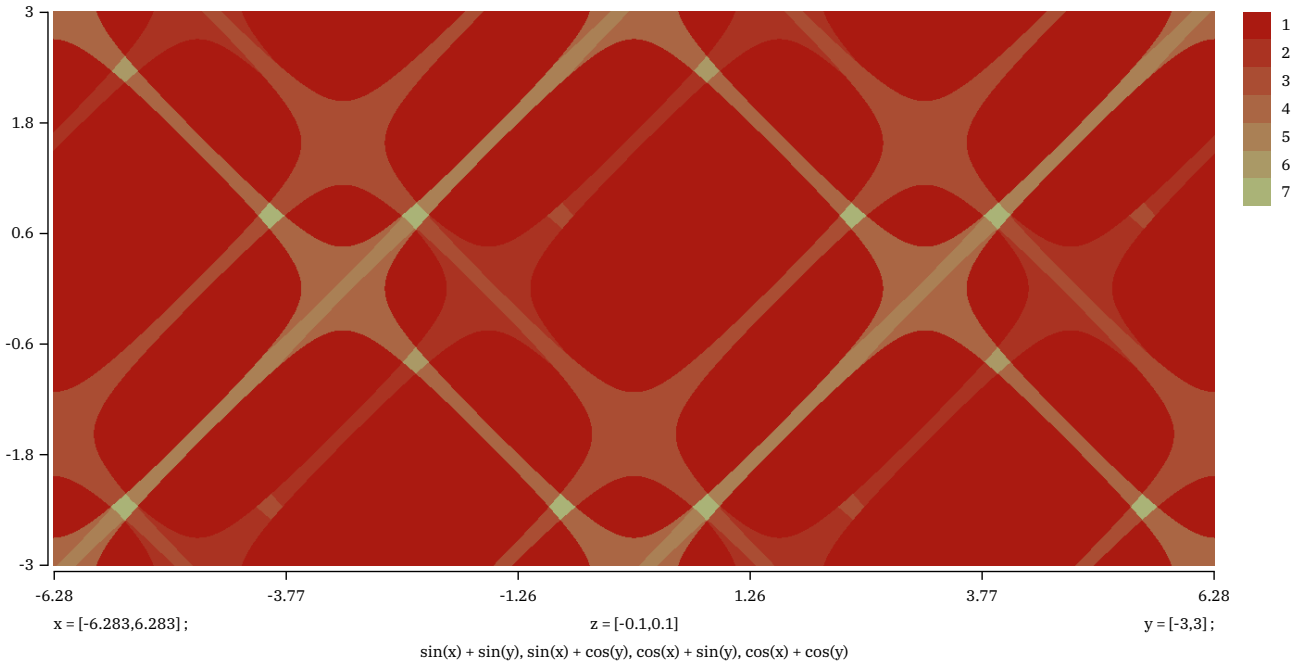


Figure 3.8

```
            "cos(x) + cos(y)"
        },
        background = "bitmap",
        foreground = "none",
        level      = 3,
        color      = "shade({1,0,0},{0,1,0})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

We can enlargen the window, which is demonstrated in figure 3.9. I suppose that such images only make sense in educational settings.

In figure 3.10 we see different combinations of backgrounds (in color) and foregrounds (edges) in action.

```
\startMPcode{doublefun}
    draw lmt_contour [
        xmin =  0, xmax = 4*pi, xstep = 0,
        ymin = -6, ymax = 6,    ystep = 0,

        levels = 5, legend = false, linewidth  = 1/2,

        preamble   = "local sin, cos = math.sin, math.cos",
        function   = "cos(x) - sin(y)",
        color      = "shade({1/2,0,0},{0,0,1/2})",

        background = "bitmap", foreground = "cell",
    ] xsized .3TextWidth ;
\stopMPcode
```

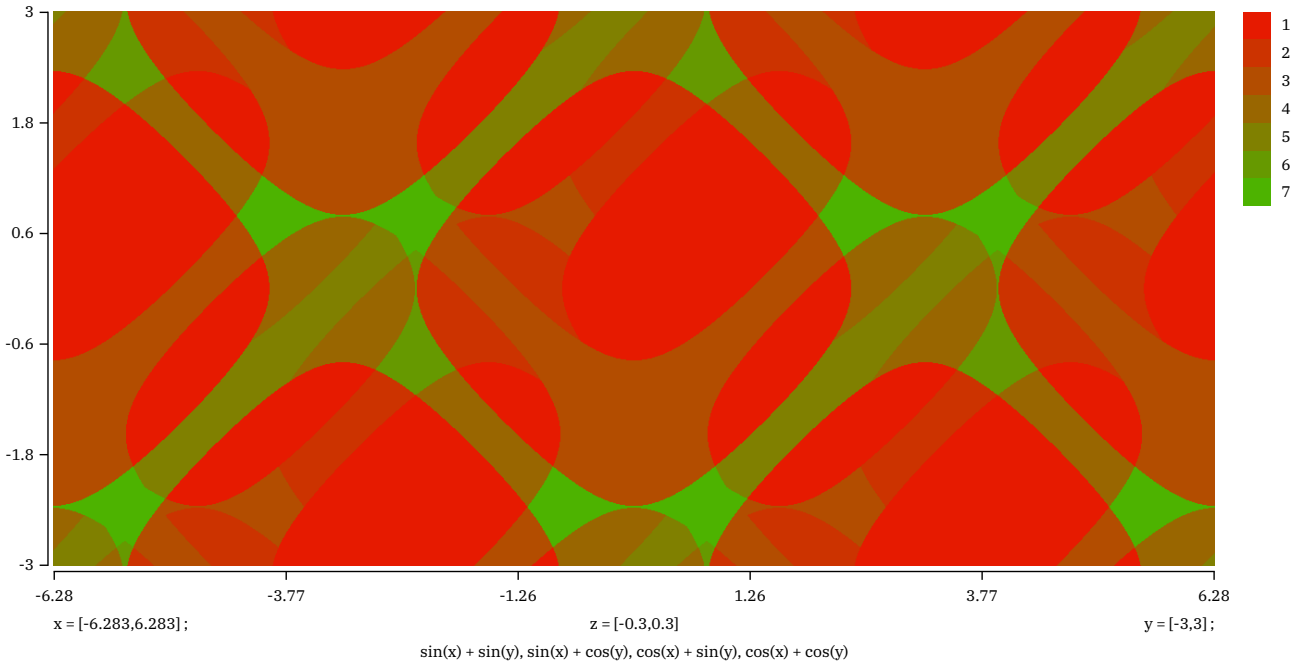
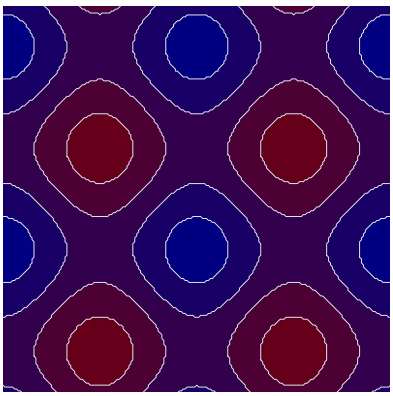


Figure 3.9

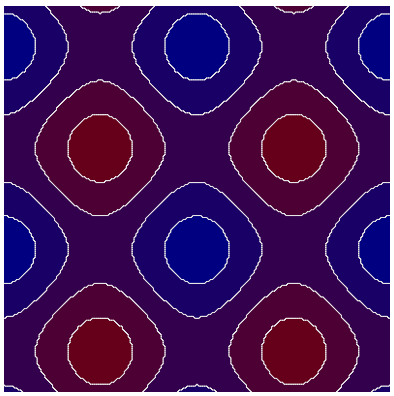
There are quite some settings. Some deal with the background, some with the foreground and quite some deal with the legend.

name	type	default	comment
xmin	numeric	0	needs to be set
xmax	numeric	0	needs to be set
ymin	numeric	0	needs to be set
ymax	numeric	0	needs to be set
xstep	numeric	0	auto 1/200 when zero
ystep	numeric	0	auto 1/200 when zero
checkresult	boolean	false	checks for overflow and NaN
defaultnan	numeric	0	the value to be used when NaN
defaultinf	numeric	0	the value to be used when overflow
levels	numeric	10	number of different levels to show
level	numeric		only show this level (foreground)
preamble	string		shortcuts
function	string	x + y	the result z value
functions	list		multiple functions (overlapping levels)
color	string	lin(l)	the result color value for level l (1 or 3 values)
colors	numeric		used when set
background	string	bitmap	band, bitmap, shape
foreground	string	auto	cell, edge, shape auto
linewidth	numeric	.25	
linecolor	string	gray	
width	numeric	0	automatic when zero
height	numeric	0	automatic when zero
trace	boolean	false	

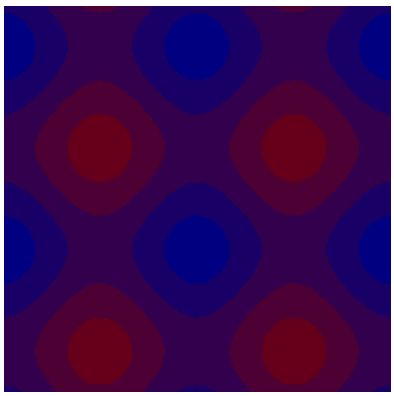
legend	string	all	x y z function range all
legendheight	numeric	LineHeight	
legendwidth	numeric	LineHeight	
legendgap	numeric	0	
legenddistance	numeric	EmWidth	
textdistance	numeric	2EmWidth/3	
functiondistance	numeric	ExHeight	
functionstyle	string		ConTEXt style name
xformat	string	@0.2N	number format template
yformat	string	@0.2N	number format template
zformat	string	@0.2N	number format template
xstyle	string		ConTEXt style name
ystyle	string		ConTEXt style name
zstyle	string		ConTEXt style name
axisdistance	numeric	ExHeight	
axislinewidth	numeric	.25	
axisoffset	numeric	ExHeight/4	
axiscolor	string	black	
ticklength	numeric	ExHeight	
xtick	numeric	5	
ytick	numeric	5	
xlabel	numeric	5	
ylabel	numeric	5	



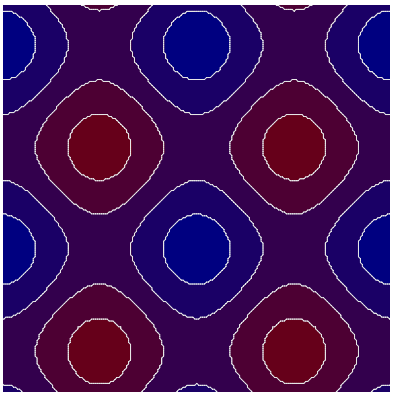
bitmap edge



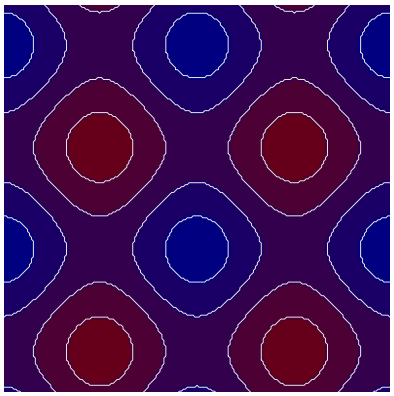
bitmap cell



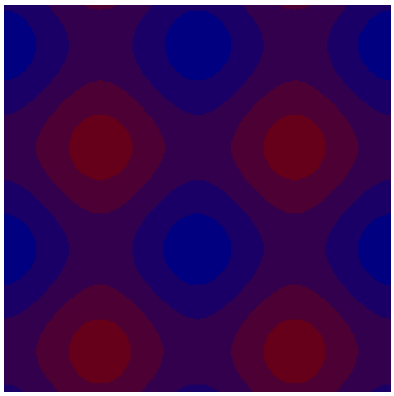
bitmap none



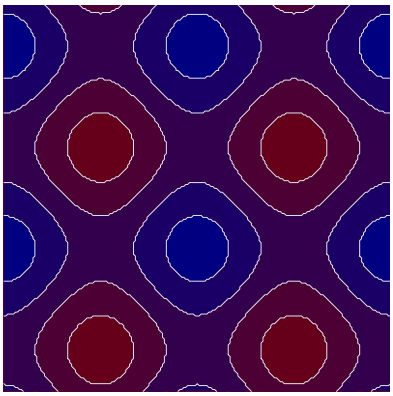
shape shape



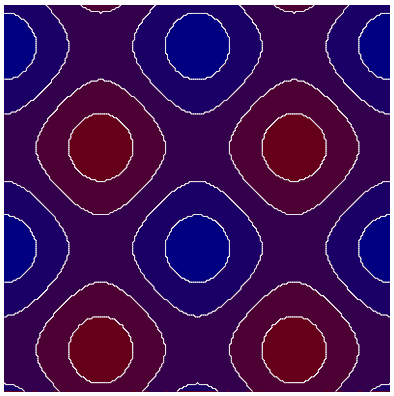
shape edge



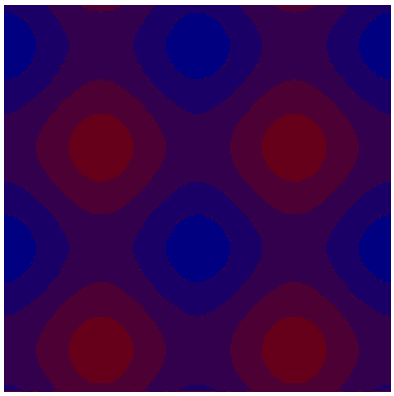
shape none



band edge



band cell



band none

Figure 3.10

4 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.

name	type	default	comment
nx	numeric	1	
dx	numeric	1	
tx	numeric	0	
sx	numeric	1	
startx	numeric	0	
ny	numeric	1	
dy	numeric	1	

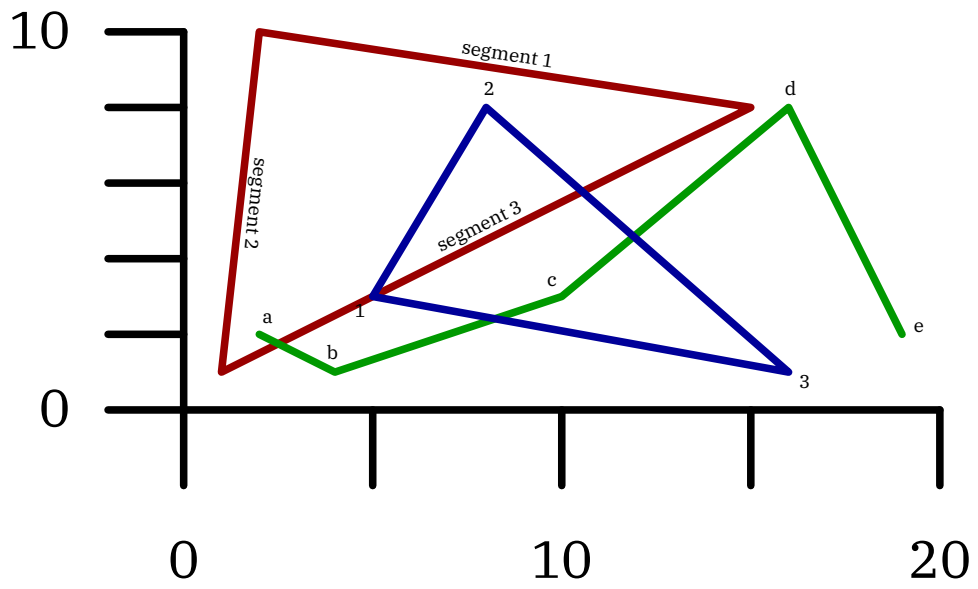


Figure 4.1

```
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
```

---

5 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 5.1 we see two examples:

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

and

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

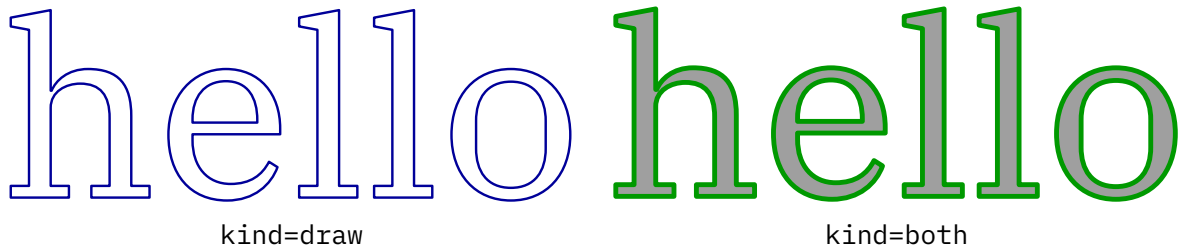


Figure 5.1 Drawing and/or filling an outline.

Normally the fill ends up below the draw but we can reverse the order, as in figure 5.2, 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
```

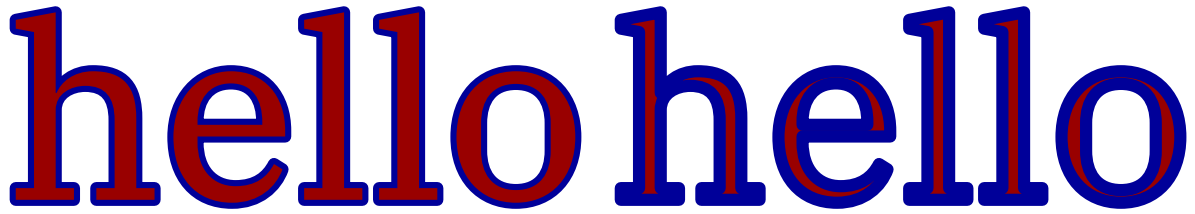


Figure 5.2 Reversing the order of drawing and filling.

It is possible to fill and draw in one operation, in which case the same color is used for both, see figure 5.3 for an example fo this. This is a low level optimization where the shape is only output once.

Figure 5.3 Combining a fill with a draw in the same color.

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

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

In this case we feed the text into the `\framed` macro so that we get a properly aligned paragraph of text, as demonstrated in figure 5.4 and ??. If you want more trickery you can of course use any ConTEXt command (including `\framed` with all kind of options) in the text.

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.

Figure 5.4 Outlining a paragraph of text.

```
\startMPcode{doublefun}
    draw lmt_outline [
        text      = "\obeydiscretionaries\samplefile{ward}",
        align     = "normal,tolerant",
        style     = "bold",
```

```
        width     = 10cm,
        kind      = "draw",
        drawcolor = "darkblue",
    ] xsized TextWidth ;
\stopMPcode
```

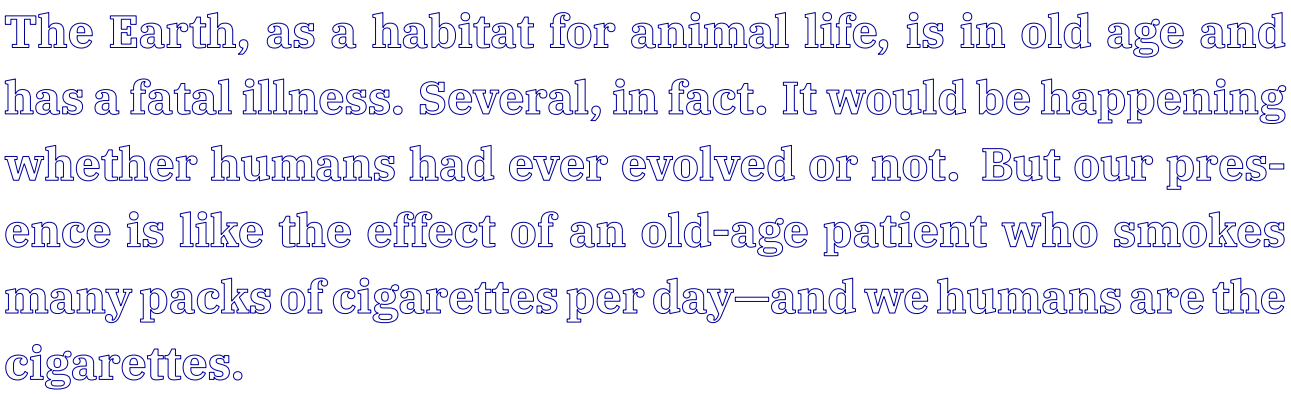


Figure 5.5 Outlining a paragraph of text with a specific width.

We summarize the parameters:

name	type	default	comment
text	string		
kind	string	draw	One of draw, fill, both, reverse and fillup.
fillcolor	string		
drawcolor	string		
rulethickness	numeric	1/10	
align	string		
style	string		
width	numeric		

6 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:

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

Here is the same example but with the text in the reverse order. The results of both examples are shown in figure 6.1.

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

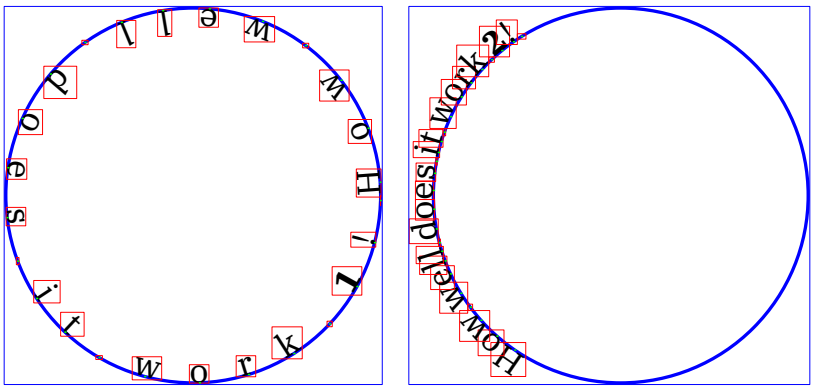


Figure 6.1

There are not that many options. One is `autoscale` which makes the shape and text match. Figure 6.2 shows what happens.

```
\startMPcode{doublefun}
    draw lmt_followtext [
        text        = "How well does it work {\bf 3}! ",
        trace       = true,
```

```
        autoscaleup = "yes"
    ] ysized 5cm ;
\stopMPcode

\startMPcode{doublefun}
    draw lmt_followtext [
        text        = "How well does it work {\bf 4}! ",
        path        = fullcircle scaled 2cm,
        trace       = true,
        autoscaleup = "max"
    ] ysized 5cm ;
\stopMPcode
```

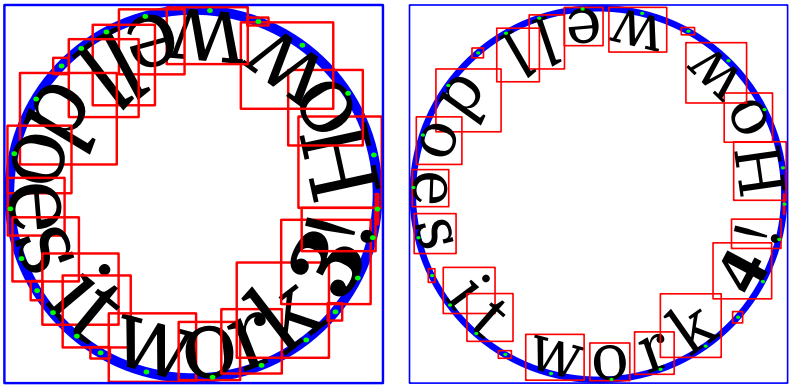


Figure 6.2

You can use quite strange paths, like the one show in figure 6.3. 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"
    ] xsized TextWidth ;
\stopMPcode
```

The small set of options is:

name	type	default	comment
text	string		
spread	string	true	
trace	numeric	false	
reverse	numeric	false	
autoscaleup	numeric	no	
autoscaledown	string	no	
path	string	(fullcircle)	

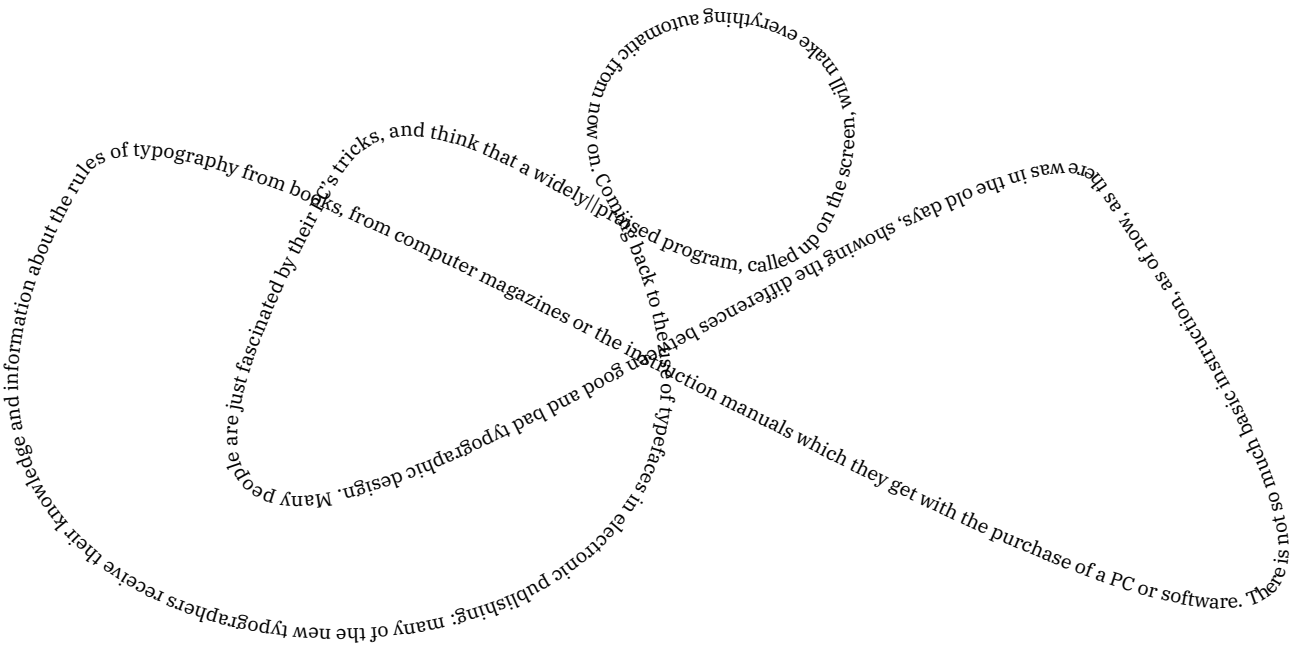

will make everything automatic from now on. Comprised program, called up on the screen,
back to the days of typefaces in electronic publishing. many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely||printed

Figure 6.3

7 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.

Figure 7.1

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

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

Figure 7.2

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

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

Figure 7.3

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

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).

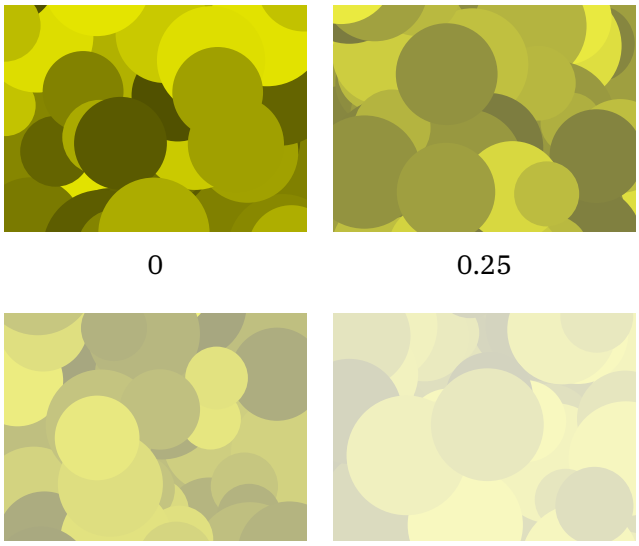


Figure 7.4

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       = 4cm,
        height      = 3cm,
        color       = "yellow",
        alternative = "circle".
        reduction   = 0.25,
    ] ;
\stopMPcode
```

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

name	type	default	comment
color	string	red	
width	numeric	1	
height	numeric	1	
reduction	numeric	0	
alternative	string	circle	

8 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 8.1 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 ;
```

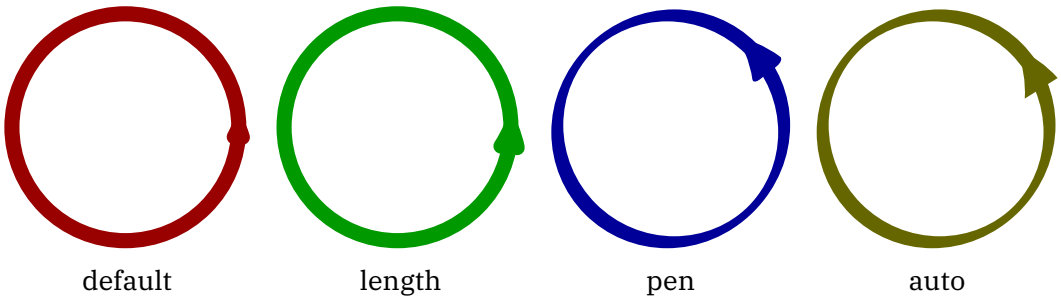
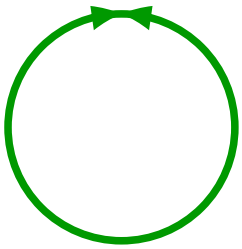


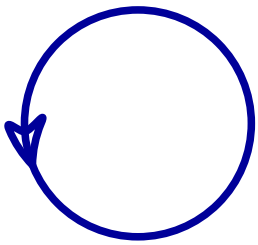
Figure 8.1

```
    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 ;
```

```
    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 ;
\stopMPcode
```

The supported parameters are:

name	type	default	comment
path	path		
pen	path		
	string	auto	
kind	string	fill	fill or draw
dimple	numeric	1/5	
scale	numeric	3/4	
penscale	numeric	3	
length	numeric	4	
angle	numeric	45	
location	string	end	end, middle or both
alternative	string	normal	normal, dimpled or curved
percentage	numeric	50	
headonly	boolean	false	

9 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 9.1).

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

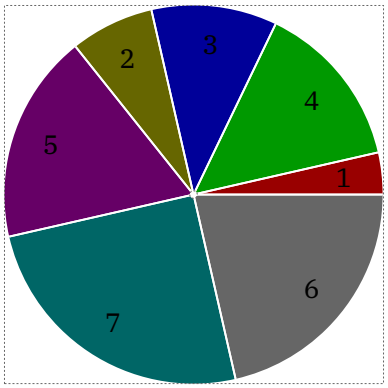


Figure 9.1

As with all these lmtx extensions, you're invited to play with the parameters. in figure 9.2 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  = "ttxx"
] ;
\stopMPcode
```

```
\startMPcode
draw lmt_chart_circle [
    height      = 4cm,
    samples     =  { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage  = false,
    trace       = true,
```

```
    linewidth   = .125mm,
    originsize  = 0,
    labeloffset = 3cm,
    labelstyle  = "bfxx",
    legendstyle = "tfxx",
    legend      = {
        "first", "second", "third", "fourth",
        "fifth", "sixths", "sevenths"
    }
] ;
```

\stopMPcode

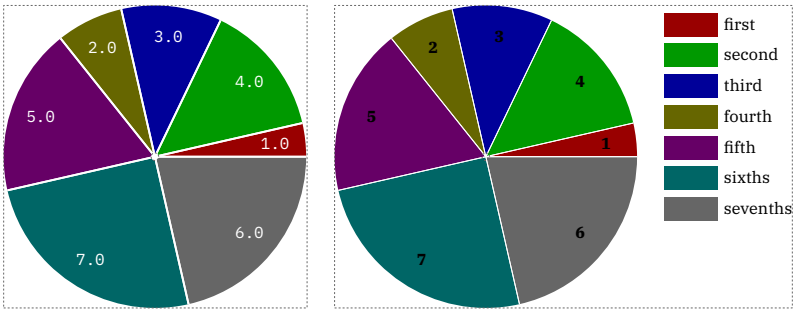


Figure 9.2

A second way of rendering are histograms, and the interface is mostly the same. In figure 9.3 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

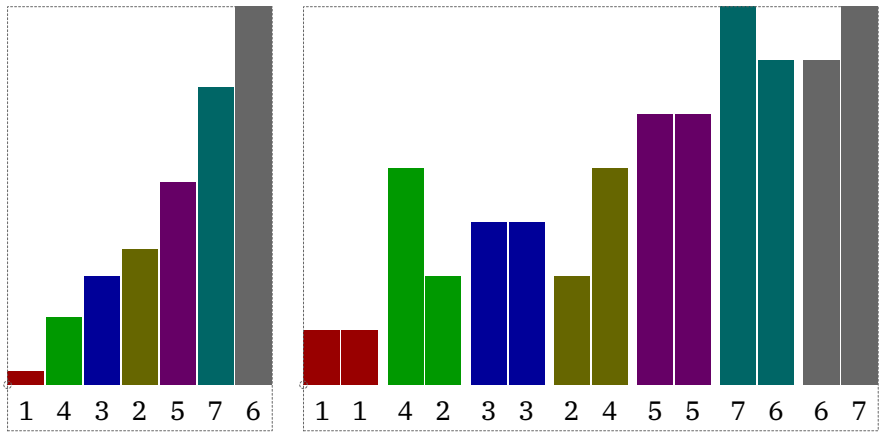


Figure 9.3

A cumulative variant is shown in figure 9.4 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
```

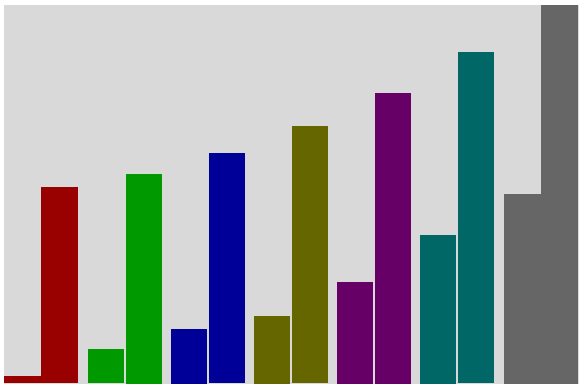


Figure 9.4

A different way of using colors is shown in figure 9.5 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 }
```

```
        },
        percentage = true,
        cumulative = true,
        showlabels = false,
        background = "lightgray",
        colormode  = "local",
    ] ;
\stopMPcode
```

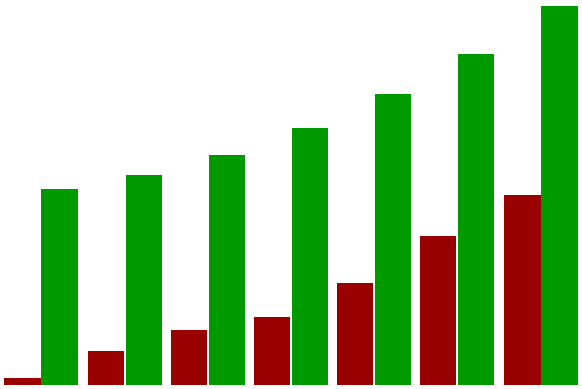


Figure 9.5

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 9.6. 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",
```

```
        labelcolor  = "black"
        labelstyle  = "bfxx"
        legendstyle = "tfxx",
        labelstrut  = "yes",
        legend      = {
            "first", "second", "third", "fourth",
            "fifth", "sixths", "sevenths"
        }
    ] ;
\stopMPcode
```

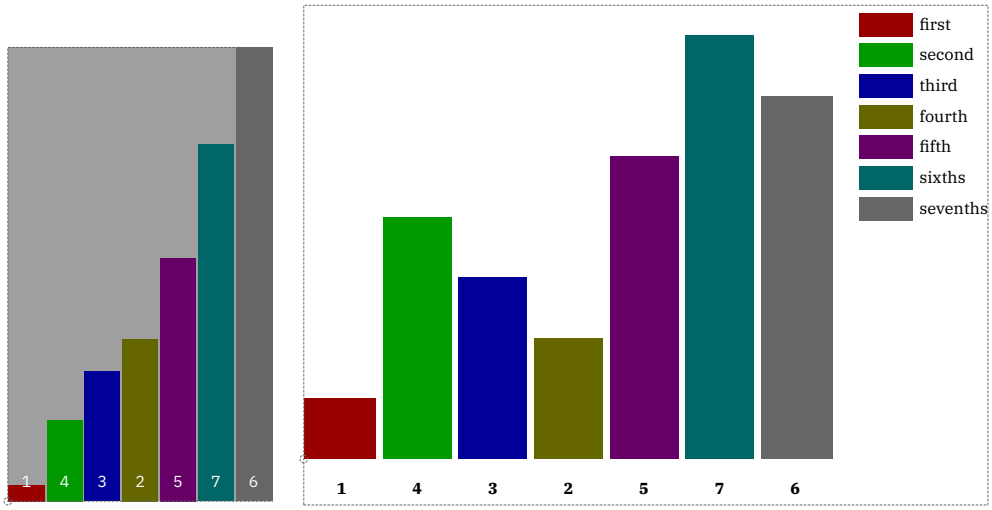


Figure 9.6

The third category concerns bar charts that run horizontal. Again we see similar options driving the rendering (figure 9.7).

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

\startMPcode
    draw lmt_chart_bar [
        samples         = { { 1, 4, 3, 2, 5, 7, 6 } },
        percentage      = true,
        cumulative      = true,
        showlabels      = false,
        backgroundcolor = "lightgray",
    ] ;
\stopMPcode
```

Determining the offset of labels is manual work:

```
\startMPcode
draw lmt_chart_bar [
    width           = 4cm,
    height          = 5mm,
    samples         = { { 1, 4, 3, 2, 5, 7, 6 } },
    percentage      = true,
    cumulative      = true,
    trace           = true,
    labelcolor      = "white",
    labelstyle      = "ttxx",
    labelanchor     = "rt",
    labeloffset     = .25EmWidth,
    backgroundcolor = "middlegray",
] ;
\stopMPcode
```

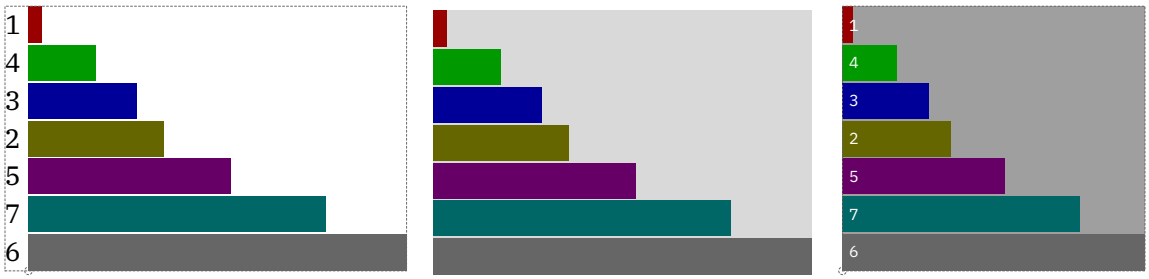


Figure 9.7

Here is one with a legend (rendered in figure 9.8):

```
\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
```

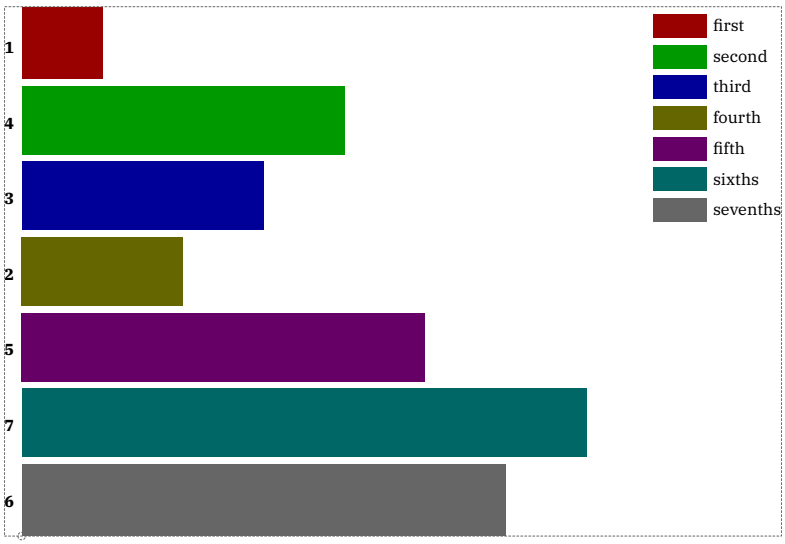


Figure 9.8

You can have labels per dataset as well as draw multiple datasets in one image, see figure 9.9:

```
\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
```

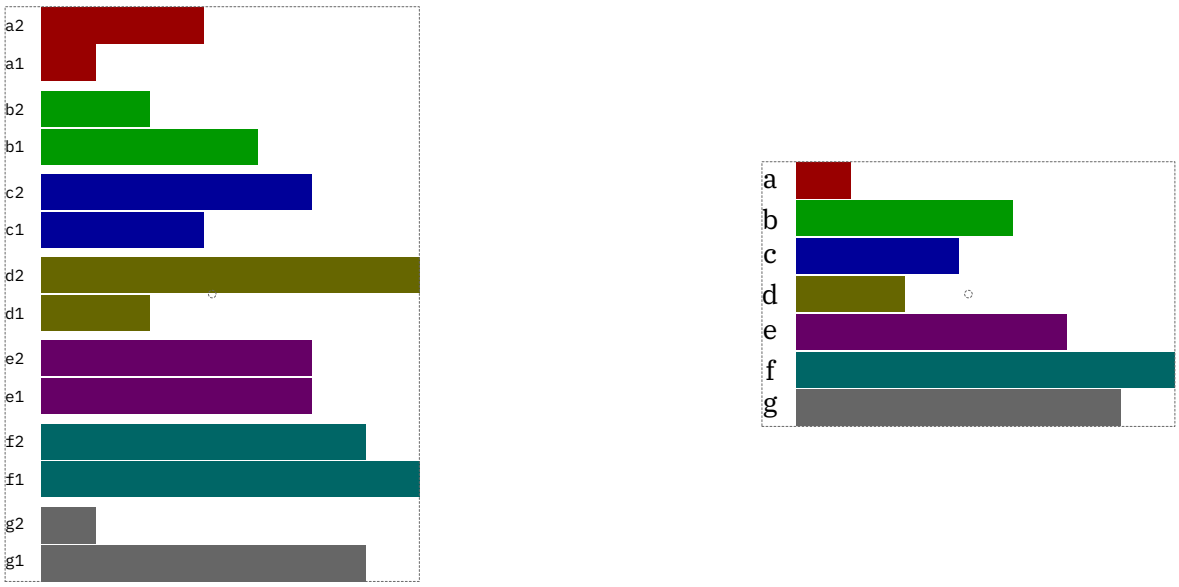


Figure 9.9

name	type	default	comment
originsize	numeric	1mm	
trace	boolean	false	
showlabels	boolean	true	
center	boolean	false	
samples	list		
	cumulative	boolean	false
percentage	boolean	false	
maximum	numeric	0	
distance	numeric	1mm	
labels	list		
labelstyle	string		
labelformat	string		
labelstrut	string	auto	
labelanchor	string		
labeloffset	numeric	0	
labelfraction	numeric	0.8	
labelcolor	string		
backgroundcolor	string		
drawcolor	string	white	
fillcolors	list		primary (dark) colors
colormode	string	global	or local
linewidth	numeric	.25mm	
legendcolor	string		
legendstyle	string		
legend	list		

Pie charts have:

name	default

height	5cm
width	5mm
labelanchor	
labeloffset	0
labelstrut	no

Histograms come with:

name	default
height	5cm
width	5mm
labelanchor	bot
labeloffset	1mm
labelstrut	auto

Bar charts use:

name	default
height	5cm
width	5mm
labelanchor	lft
labeloffset	1mm
labelstrut	no

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
  [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}
    fill OverlayBox withcolor "darkyellow" ;
    save p ; path p[] ;
    p1 := unitsquare xysized( OverlayWidth/4, OverlayHeight/4) ;
    p2 := unitsquare xysized(2OverlayWidth/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) :
```

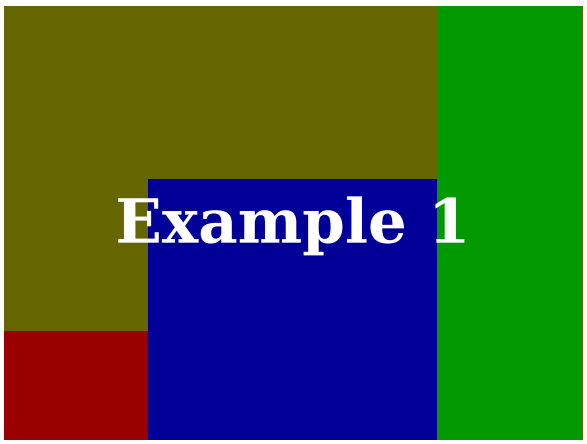


Figure 10.1

```
        (center q) -- (point (i-1) of q) -- (point i of q) -- (center q) --
    endfor cycle ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        paths = { p }
    ] withcolor "darkred" ;

    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath3}
    save q ; path q ; q := unitcircle xysized(OverlayWidth,OverlayHeight)
      randomized 3mm ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        paths = { meshed(q,OverlayBox,.05) }
    ] withcolor "darkgreen" ;
  % draw OverlayMesh(q,.025) withcolor "darkgreen" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{MyPath4}
    save q ; path q ; q := unitcircle xysized(OverlayWidth,OverlayHeight)
      randomized 3mm ;
    fill q withcolor "darkgray" ;
    draw lmt_mesh [
        trace = true,
        auto  = true,
        step  = 0.0125,
        paths = { q }
    ] withcolor "darkyellow" ;
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic

\startuseMPgraphic{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(2OverlayWidth/5,2OverlayHeight/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(2OverlayWidth/5,2OverlayHeight/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" ;
```

```
    setbounds currentpicture to OverlayBox ;
\stopuseMPgraphic
```

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.

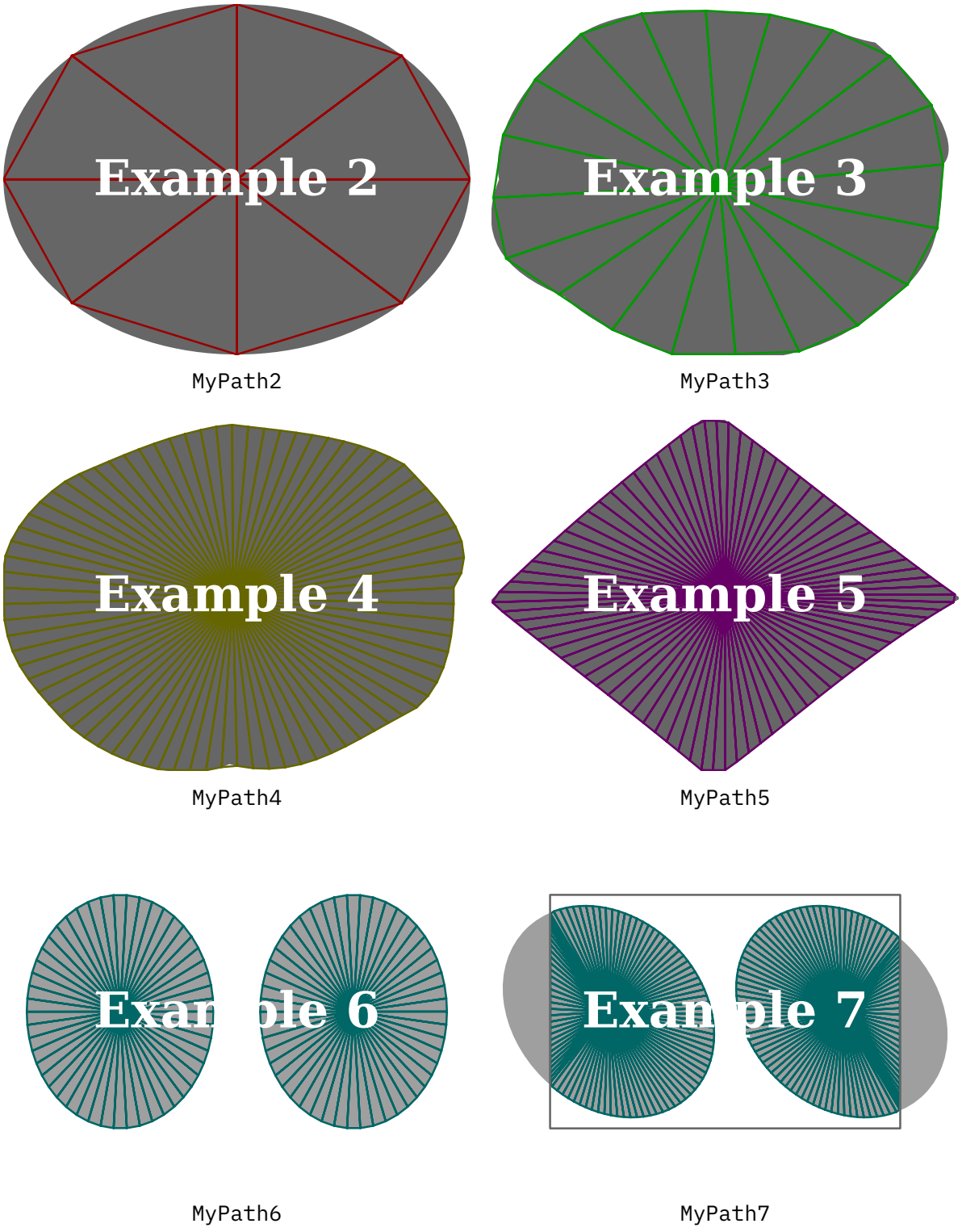


Figure 10.2

11 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 that 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 11.1 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",
```

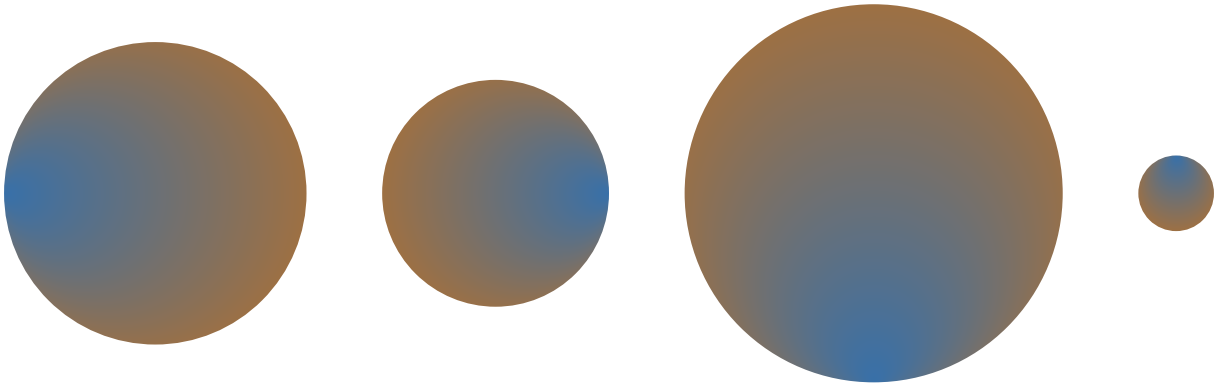


Figure 11.1 Simple circular shades.

```
    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 11.2 Simple rectangular shades.

The direction relates to the boundingbox. Instead of a keyword you can also give two values, indicating points on the boundingbox. Because a boundingbox 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.

```
\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
```

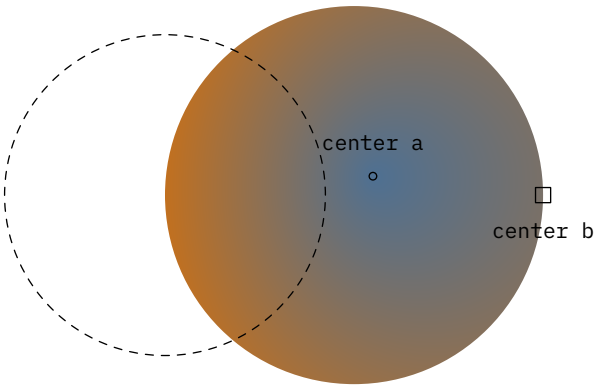


Figure 11.3 Moving the centers.

A vector takes the given points on the path as centers for the colors, see figure 11.4.

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

Messing with the radius in combination with the previously mentioned domain is really trial and error, as seen in figure 11.5.

```
\startMPcode
draw lmt_shade [
```

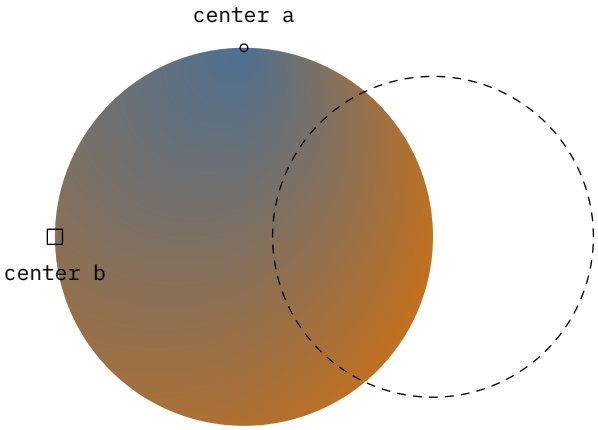


Figure 11.4 Using a vector (points).

```
    path      = fullcircle scaled 5cm,
    domain    = { 0.5, 2.5 },
    radius    = { 2cm, 6cm },
    direction = "right",
    colors    = { "MyColor3", "MyColor4" },
    trace     = true,
] ;
\stopMPcode
```

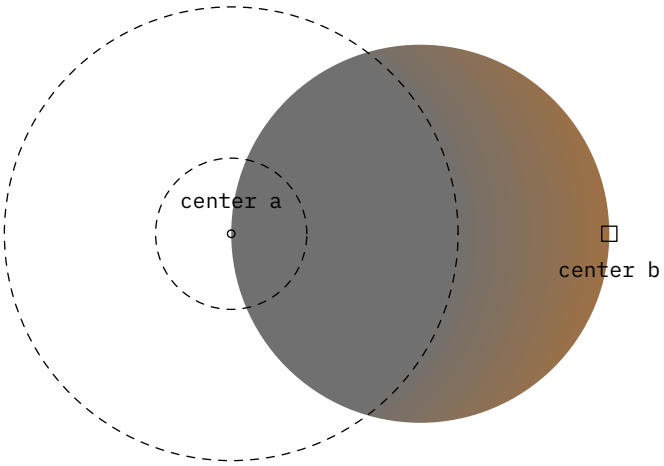


Figure 11.5 Tweaking the radius.

But actually the radius used alone works quite well as shown in figure 11.6.

```
\startMPcode
draw lmt_shade [
    path        = fullcircle scaled 5cm,
    colors      = { "red", "green" },
    trace       = true,
] ;

draw lmt_shade [
    path        = fullcircle scaled 5cm,
    colors      = { "red", "green" },
    radius      = 2.5cm,
```

```
    trace       = true,
] shifted (6cm,0) ;

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

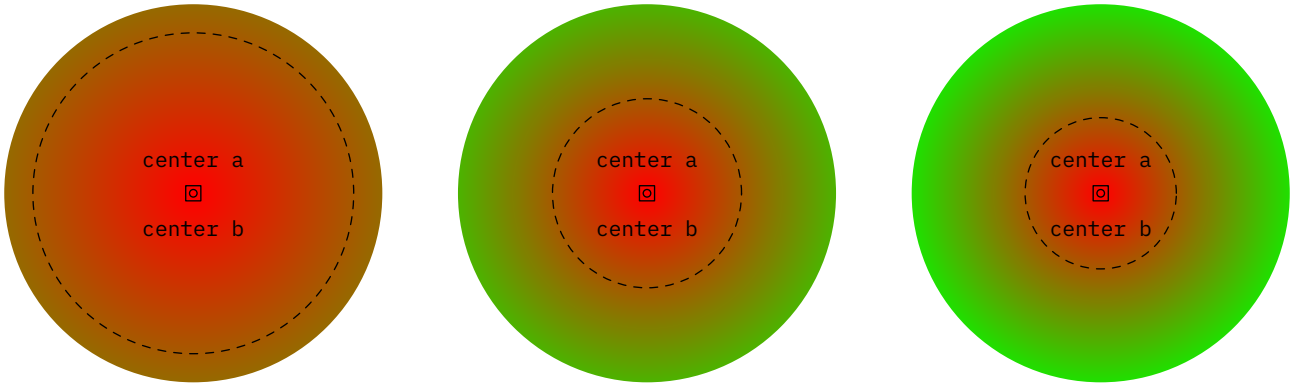


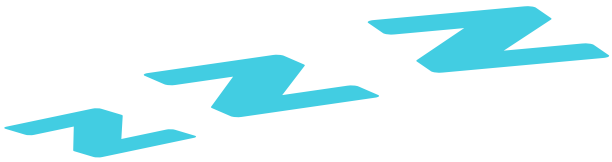
Figure 11.6 Just using the radius.

name	type	default	comment
alternative	string	circular	or linear
path	path		
trace	boolean	false	
domain	set of numerics		
radius	numeric		
	set of numerics		
factor	numeric		
origin	pair		
	set of pairs		
vector	set of numerics		
colors	set of strings		
center	numeric		
	set of numerics		
direction	string		up, down, left, right
	set of numerics		two points on the boundingbox

12 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
    draw lmt_svg [
        filename = "mozilla-svg-002.svg",
        height   = 2cm,
        width    = 8cm,
    ] ;
\stopMPcode
```



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

```
\startMPcode
    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);
\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:

name	type	default	comment
filename	path		
width	numeric		
height	numeric		

13 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 wapper 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:

`presetparameters`	`name [...]`	Assign default values to a category of parameters. Sometimes it makes sense not to set a default, because then you can check if a parameter has been set at all.
`applyparameters`	`name macro`	This prepares the parameter handler for the given category and calls the given macro when that is done.
`getparameters`	`name [...]`	The parameters given after the category name are set.
`hasparameter`	`names`	Returns `true` when a parameter is set, and `false` otherwise.
`hasoption`	`names options`	Returns `true` when there is overlap in given options, and `false` otherwise.
`getparameter`	`names`	Resolves the parameter with the given name. because a parameter itself can have a parame-

		ter list you can pass additional names to reach the final destination.
getparameterdefault	names	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.
getparametercount	names	Returns the size if a list (array).
getmaxparametercount	names	Returns the size if a list (array) but descends into lists to find the largest size of a sublist.
getparameterpath	names string boolean	Returns the parameter as path. The optional string is one of `--`, `..` or `...` and the also optional boolean will force a closed path.
getparameterpen	names	Returns the parameter as pen (path).
getparametertext	names boolean	Returns the parameter as string. The boolean can be used to force prepending a so called `\strut`.
pushparameters	category	Pushed the given (sub) category onto the stack so that we don't need to give the category each time.
popparameters		Pops the current (sub) category from the stack.

Most commands accept a list of strings separated by one or more spaces, The resolved will then stepwise 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
```