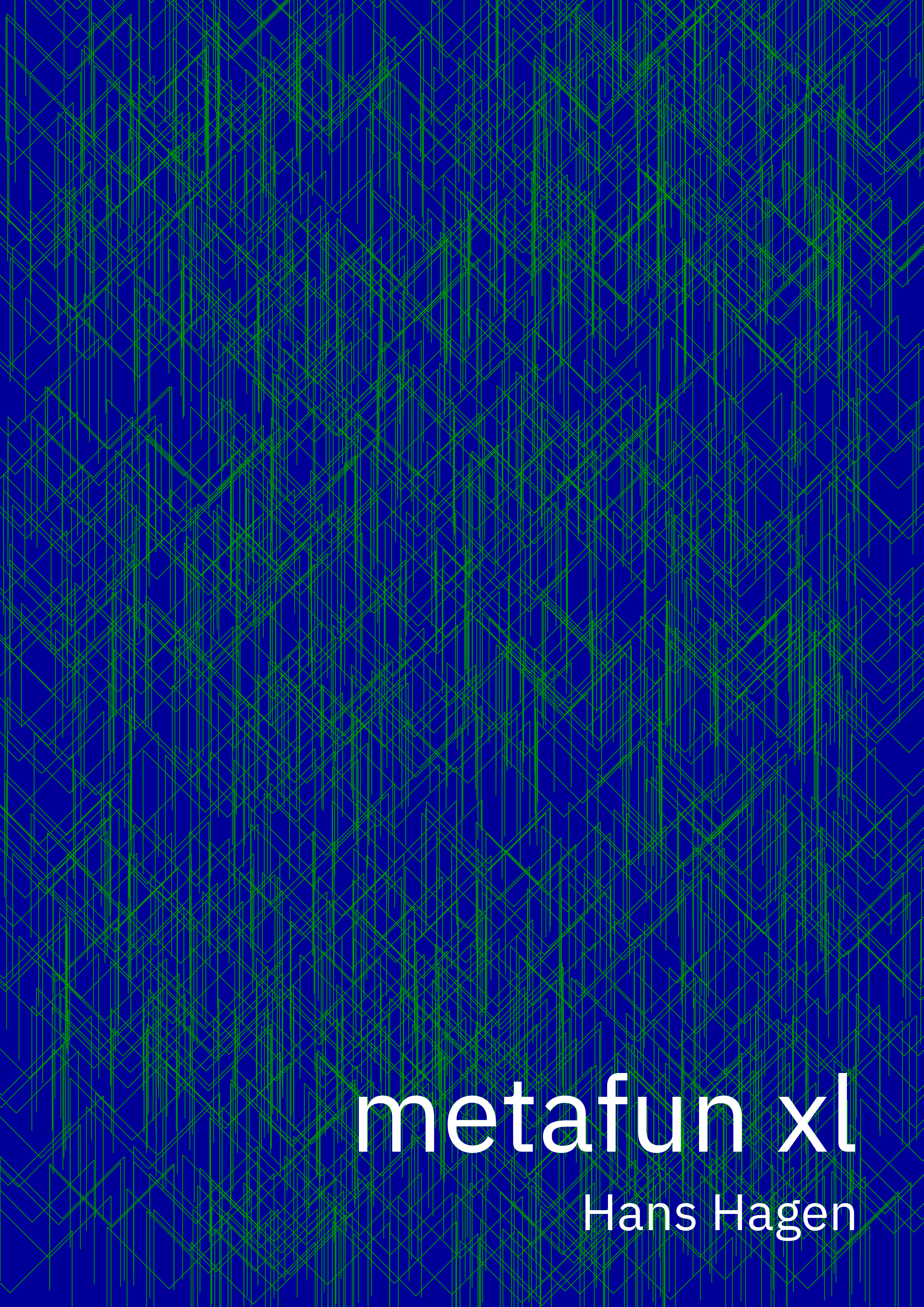


metafun xl

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 2021 (and beyond)

1 Technology

The MetaPost library that we use in LuaMetaTEX is a follow up on the library used in LuaTEX which itself is a follow up on the original MetaPost program that again was a follow up on Don Knuths MetaFont, the natural companion to TEX.

When we start with John Hobbies MetaPost we see a graphical engine that provides a simple but powerful programming language meant for making graphics, not the freehand kind, but the more systematic ones. The output is PostScript but a simple kind that can easily be converted to pdf.[1] It's output is very accurate and performance is great.

As part of the LuaTEX development project Taco Hoekwater turned MetaPost into mplib, a downward compatible library where MetaPost became a small program using that library. But there is more: there are (when enabled) backends that produce png or svg, but when used these also add dependencies on moving targets. The library by default uses the so called scaled numbers: floats that internally are long integers. But it can also work in doubles, decimal and binary and especially the last two create a dependency on libraries. It is good to notice that as in the original MetaPost the PostScript output handling is visible all over the source. Also, the way Type1 fonts are handled has been extended, for instance by providing access to shapes.

At some point a Lua interface got added that made it possible to call out to the Lua instance used in LuaTEX, so the three concepts: TEX, MetaPost and Lua can combine forces. A snippet of code can be run, and a result can be piped back. Although there is some limited access to MetaPost internals, the normal way to go is by serializing MetaPost data to the Lua end and let MetaPost scan the result using `scantokens`.

The library in LuaMetaTEX is a bit different. Of course it has the same core graphic engine, but there is no longer a backend. In ConTEXt MkIV the PostScript (and other) backends were not used anyway because it operates on the exported Lua representation of the result. Combined with the `prescript` and `postscript` features introduced in the library that provides all we need to make interesting extensions to the graphical engine (color, shading, image inclusion, text, etc). The MetaPost font support features are also not used because we need support for OpenType and even in MkII (for pdfTEX and XƎTEX) we used a different approach to fonts.

It is for that reason that the library we use in LuaMetaTEX is a leaner version of its ancestor. As mentioned, there is no backend code, only the Lua export, which saves a lot, and there are no traces of font support left, which also drops many lines of code. We forget about the binary number model because it needs a large library that also occasionally changes, but one can add it if needed. This means that there are no dependencies except for decimal but that library is relatively small and doesn't change at all. It also means that the resulting mplib library is much smaller, but it's still a substantial component in LuaMetaTEX. Internally I use the future version number 3. The original MetaPost program is version 1, so the library got version 2, and that one basically being frozen (it's in bug-fix mode) means that it will stick to that.

Another difference is that from the Lua end one has access to several scanners and also has possibilities to efficiently push back results to the engine. Running scripts can also be done more efficient. This

---

[1] For that purpose I wrote a converter in the TEX language for pdfTEX, and even within the limitations of TEX at that time (fonts, number of registers, memory) it worked out quite well.

permits a rather efficient (in terms of performance and memory usage) way to extend the language and add for instance key/value based interfaces. There are some more additions, like for instance pre- and postscripts to clip, boundary and group objects. Internals can be numeric, string and boolean. One can use utf input although that has also be added to the ancestor. Some redundant internal input/output remapping has been removed and we are more tolerant to newlines in return values from Lua. Error messages have been normalized, internal documentation cleaned up a bit. A few anomalies have been fixed too. All in- and output is now under Lua control. Etcetera. The (now very few) source files are still cweb files but the conversion to C is done with a Lua script that uses (surprise) the LuaMetaTEX engine as Lua processor. This give a bit nicer C output for when we view it in e.g. Visual Studio too (normally the cweb output is not meant to be seen by humans).

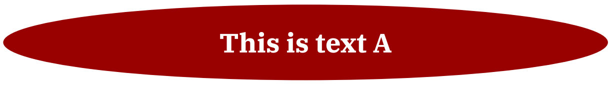
Keep in mind that it's still MetaPost with all it provided, but some has to be implemented in macros or in Lua via callbacks. The simple fact that the original library is the standard and is also the core of MetaPost most of these changes and additions cannot be backported to the original, but that is no big deal. The advantage is that we can experiment with new features without endangering users outside the ConTEXt bubble. The same is true for the Lua interface, which already is upgraded in many aspects.

2 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
```

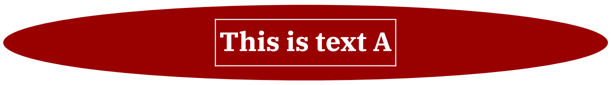
We get:



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
```

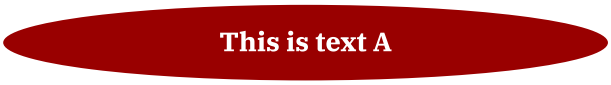
Of course you can as well draw a frame in MetaPost but the `\framed` command has more options, like alignments.



Here is a variant using the MetaFun interface:

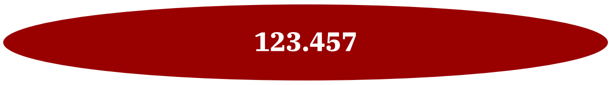
```
\startMPcode
    fill fullcircle xyscaled (8cm,1cm) withcolor "darkred" ;
    draw lmt_text [
        text  = "This is text A",
        color = "white",
        style = "bold"
    ] ;
\stopMPcode
```

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
```

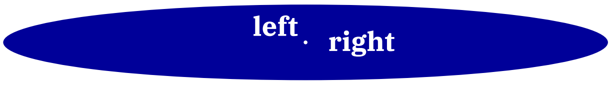


The following parameters can be set:

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
```



3 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	

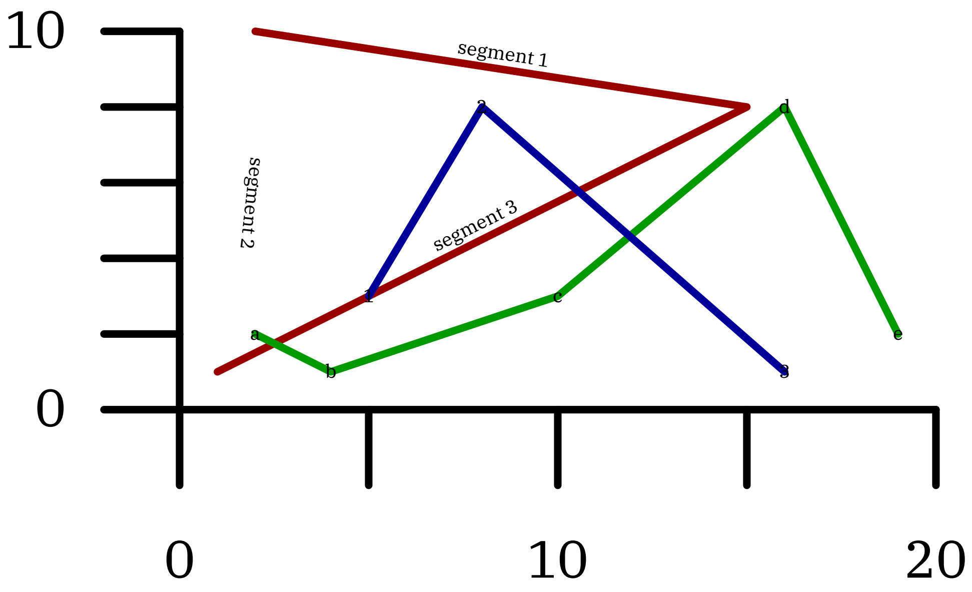


Figure 3.1

`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`	

4 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 4.1 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
```

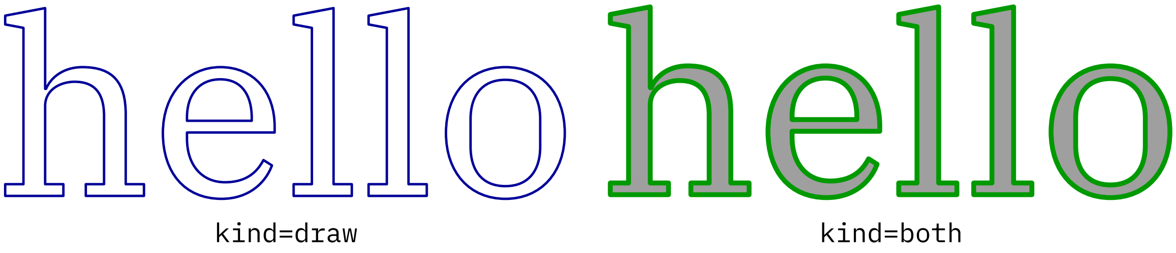
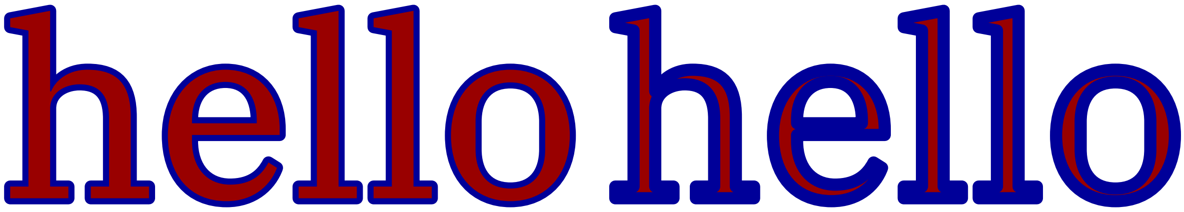


Figure 4.1 Drawing and/or filling an outline.

Normally the fill ends up below the draw but we can reverse the order, as in figure 4.2, where we coded the leftmost example as:

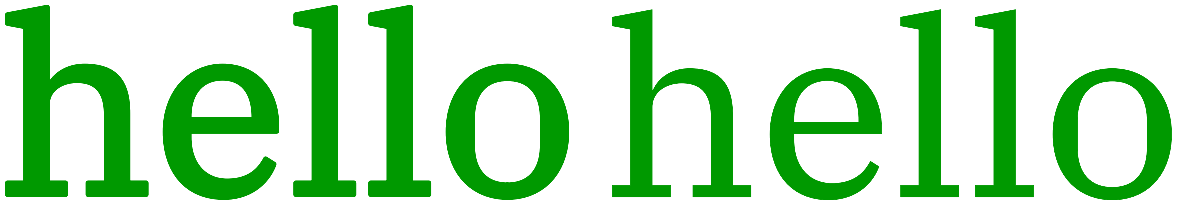
```
\startMPcode{doublefun}
    draw lmt_outline [
        text          = "hello",
        kind          = "reverse",
        fillcolor     = "darkred",
        drawcolor     = "darkblue",
        rulethickness = 1/2,
    ] xsized .45TextWidth ;
\stopMPcode
```



kind=reverse kind=both

Figure 4.2 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 4.3 for an example fo this. This is a low level optimization where the shape is only output once.



kind=fillup kind=fill

Figure 4.3 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 4.4 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 4.4 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
```

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 4.5 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`		

5 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 5.1.

```
\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
```

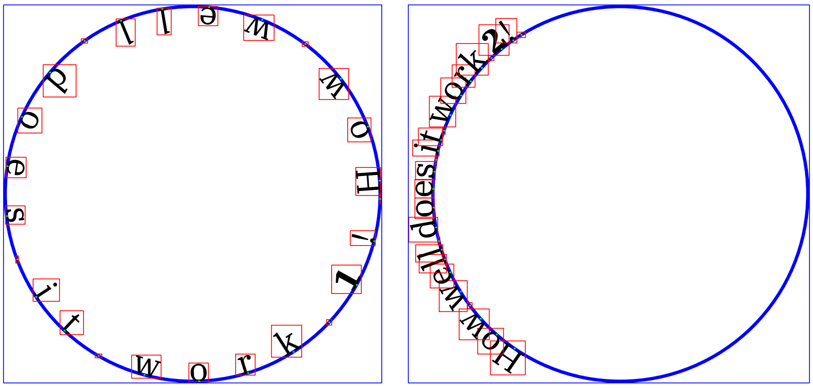


Figure 5.1

There are not that many options. One is `autoscale` which makes the shape and text match. Figure 5.2 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
```

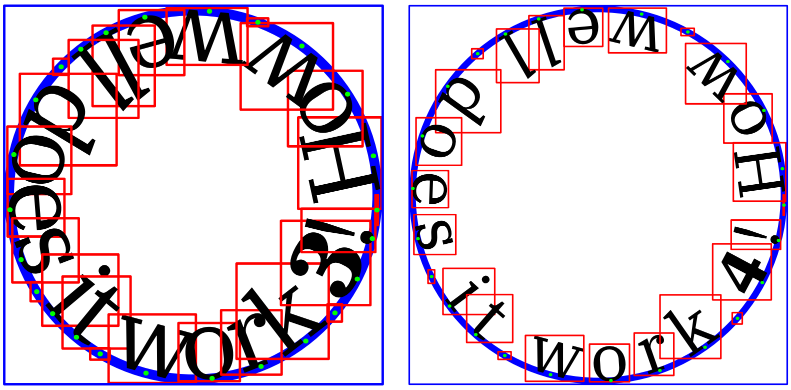


Figure 5.2

You can use quite strange paths, like the one show in figure 5.3. 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)`	

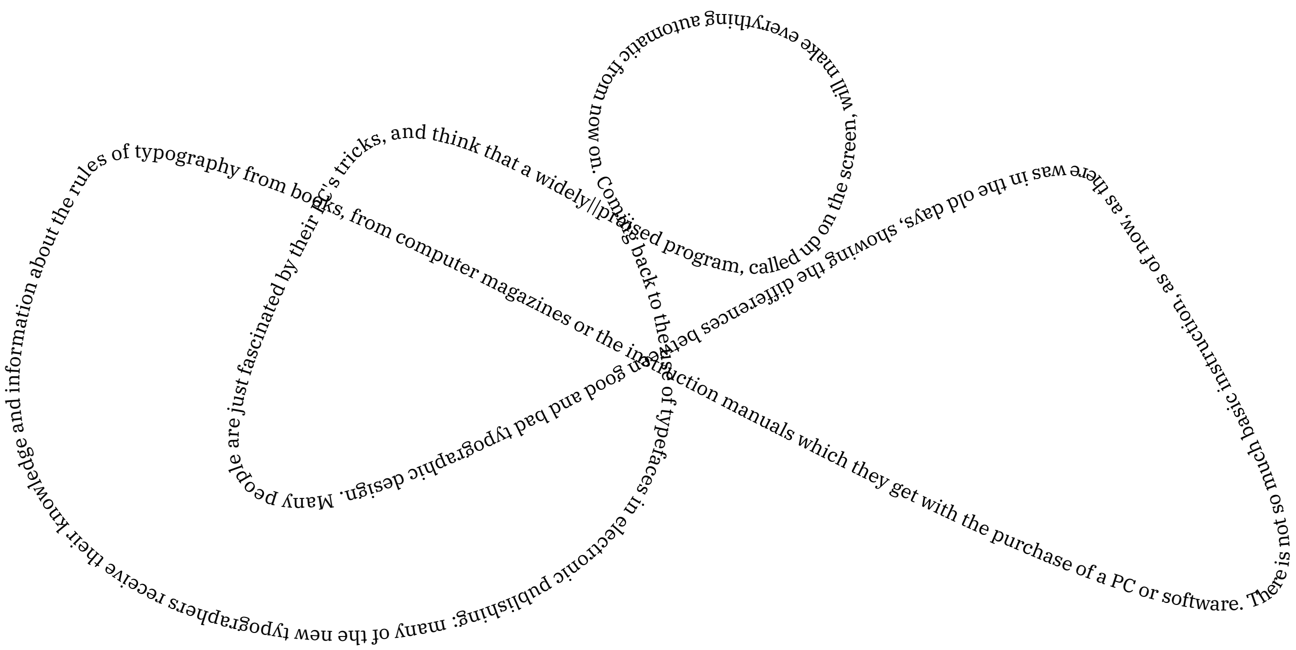

Many people are just fascinated by their PC's tricks, and think that a widely||promised program, called up on the screen, will make everything automatic from now on. Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about 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.

Figure 5.3

6 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.

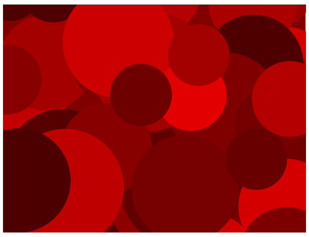


Figure 6.1

```
\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.

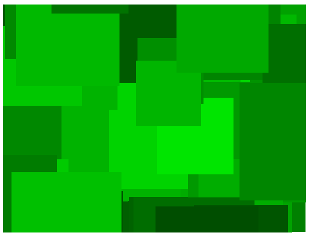


Figure 6.2

```
\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.

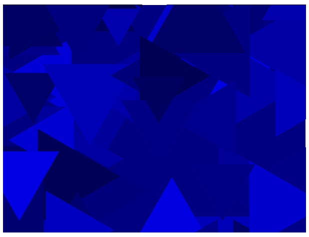


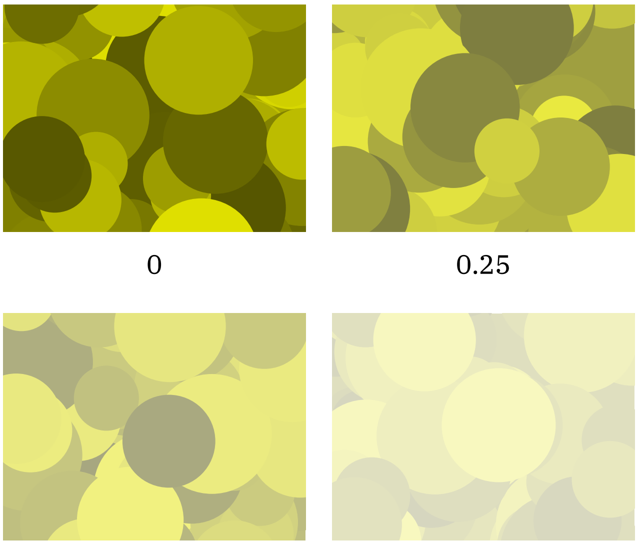
Figure 6.3

```
\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).

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
```



0 0.25

0.50 0.75

Figure 6.4

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`	

7 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 7.1 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 145),
    pen  = (pencircle xscaled 4mm yscaled 2mm rotated 45),
]
    withpen pencircle xscaled 1mm yscaled .5mm 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 ;
```

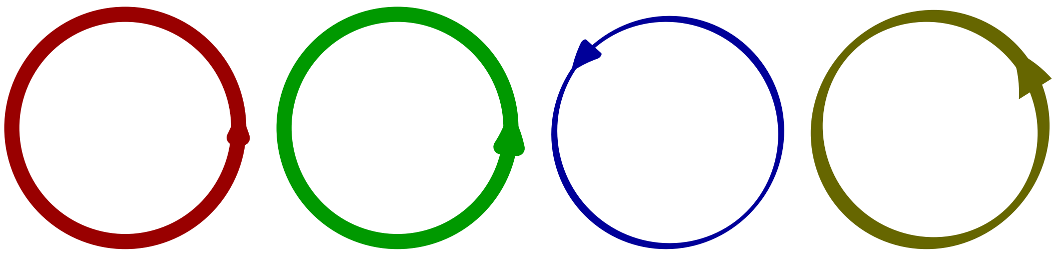
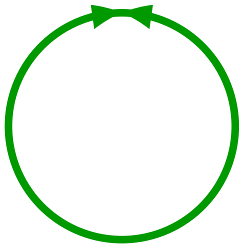

default length pen auto

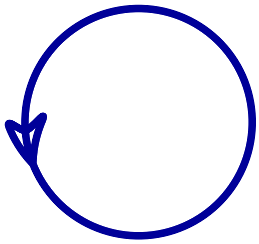
Figure 7.1

```
    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:

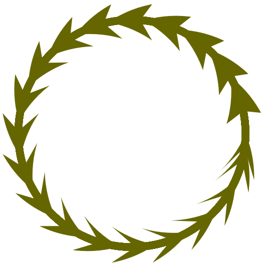
```
\startMPcode
    pickup pencircle scaled 1mm ;
    draw lmt_arrow [
        kind        = "draw",
        pen         = "auto",
        penscale    = 4,
        location    = "middle",
        alternative = "curved",
        path        = fullcircle scaled 3cm,
    ] withcolor "darkblue" ;
\stopMPcode
```



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	

8 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 8.1 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",
```

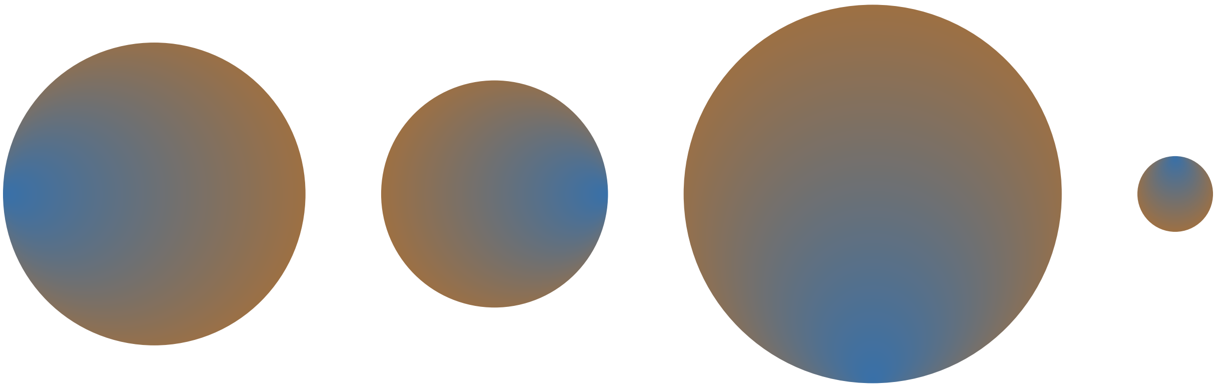


Figure 8.1 Simple circular shades.

```
    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
```

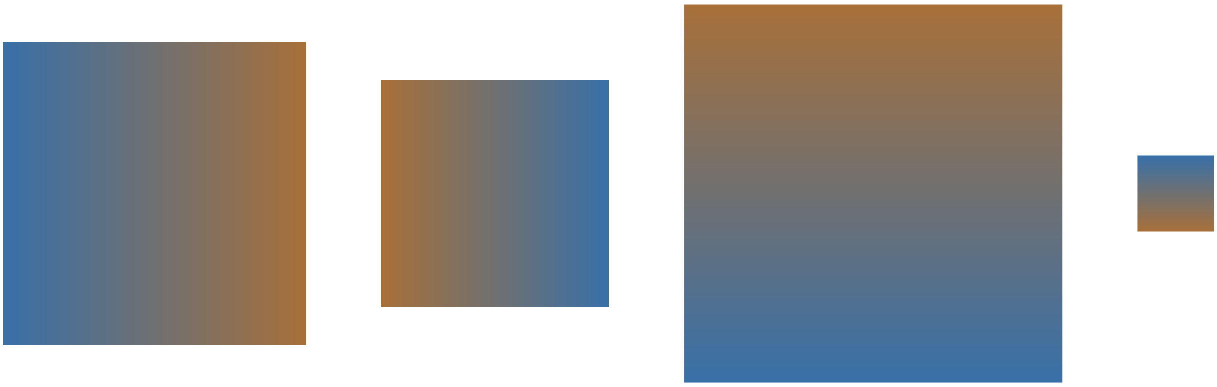


Figure 8.2 Simple rectangular shades.

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
```

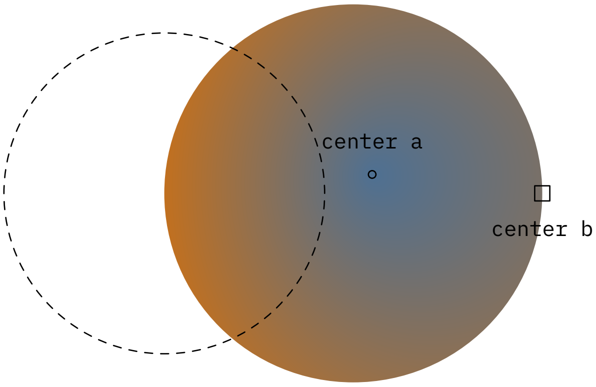


Figure 8.3 Moving the centers.

A vector takes the given points on the path as centers for the colors, see figure 8.4.

```
\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 8.5.

```
\startMPcode
draw lmt_shade [
```

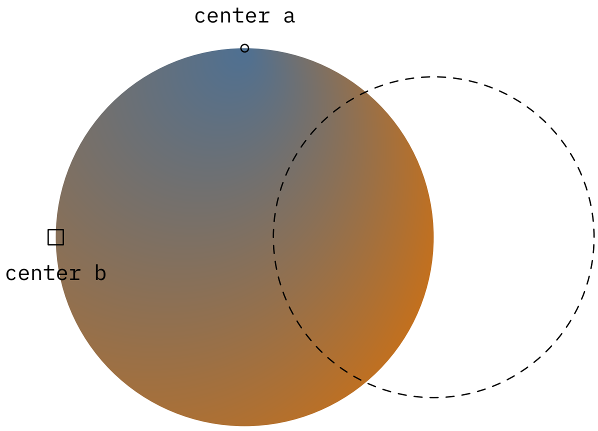


Figure 8.4 Using a vector (points).

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

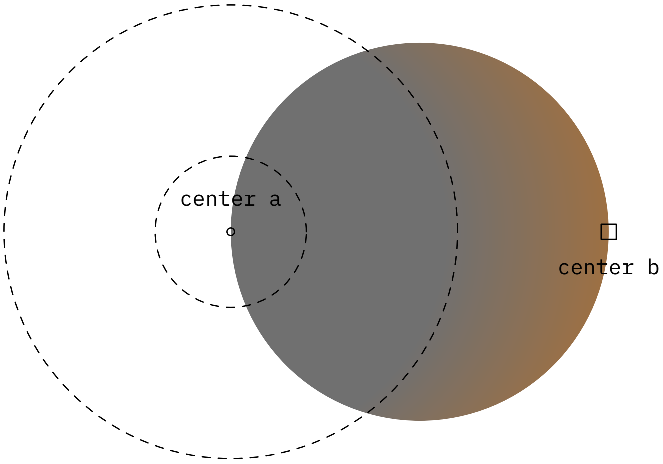


Figure 8.5 Tweaking the radius.

But actually the radius used alone works quite well as shown in figure 8.6.

```
\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
```

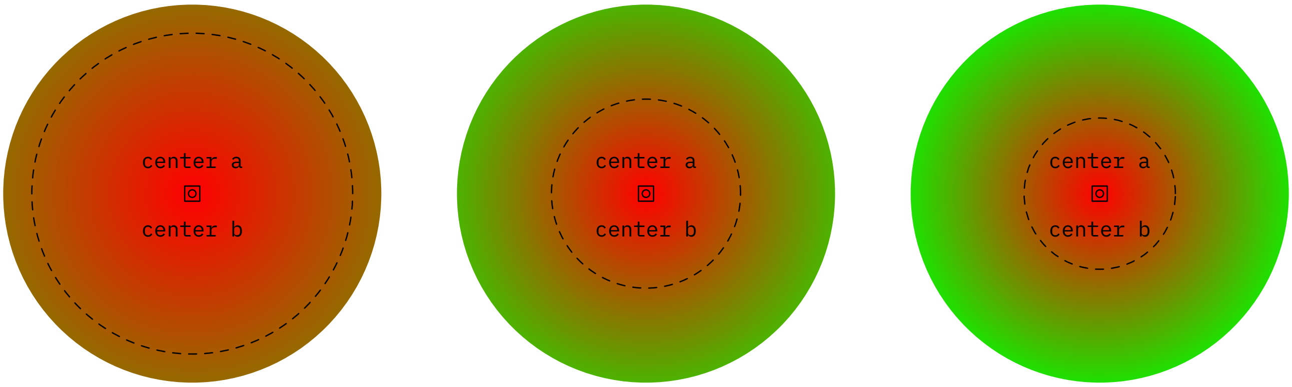


Figure 8.6 Just using the radius.

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

9 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[2]. 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",
```

---

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

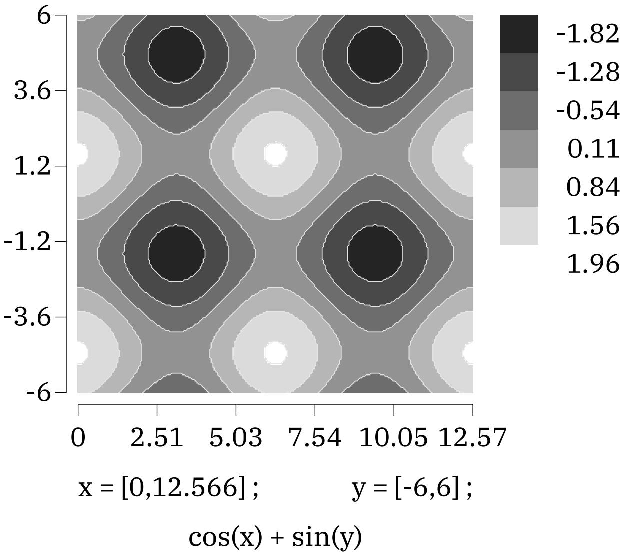


Figure 9.1

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

In figure 9.1 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 9.2 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 9.3. 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.

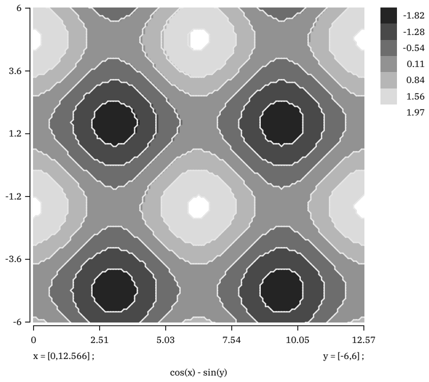
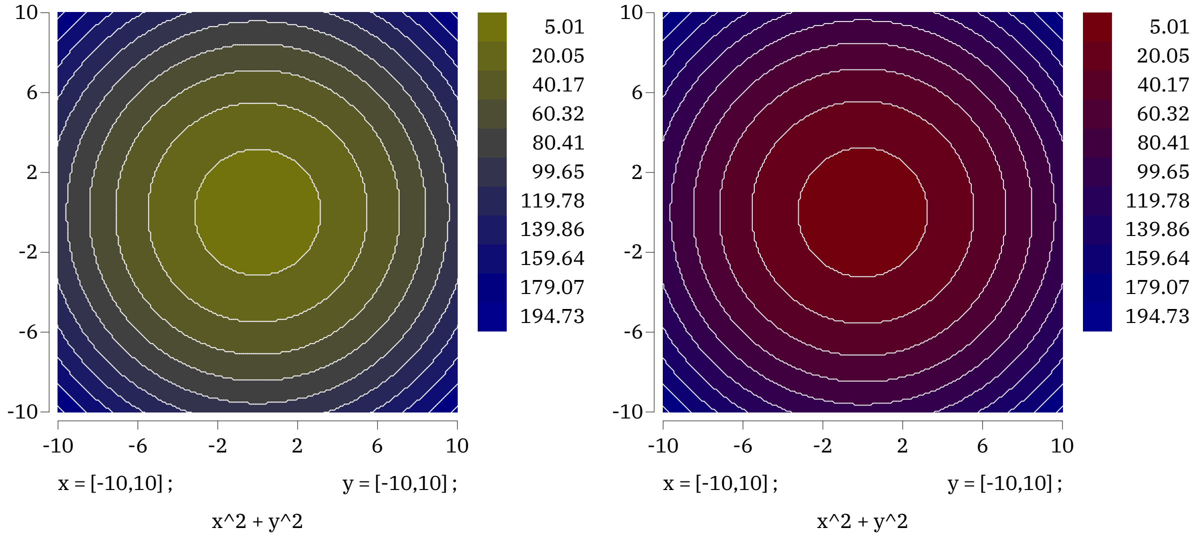


Figure 9.2

```
\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
```



shape

bitmap

Figure 9.3

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
```

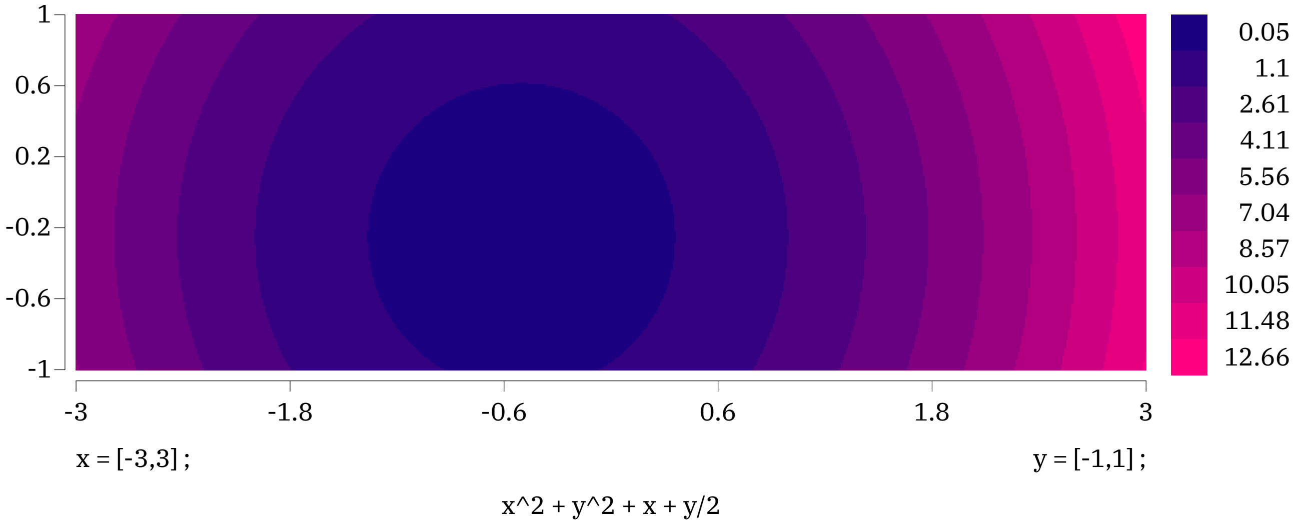


Figure 9.4

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 9.5.

```
\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
```

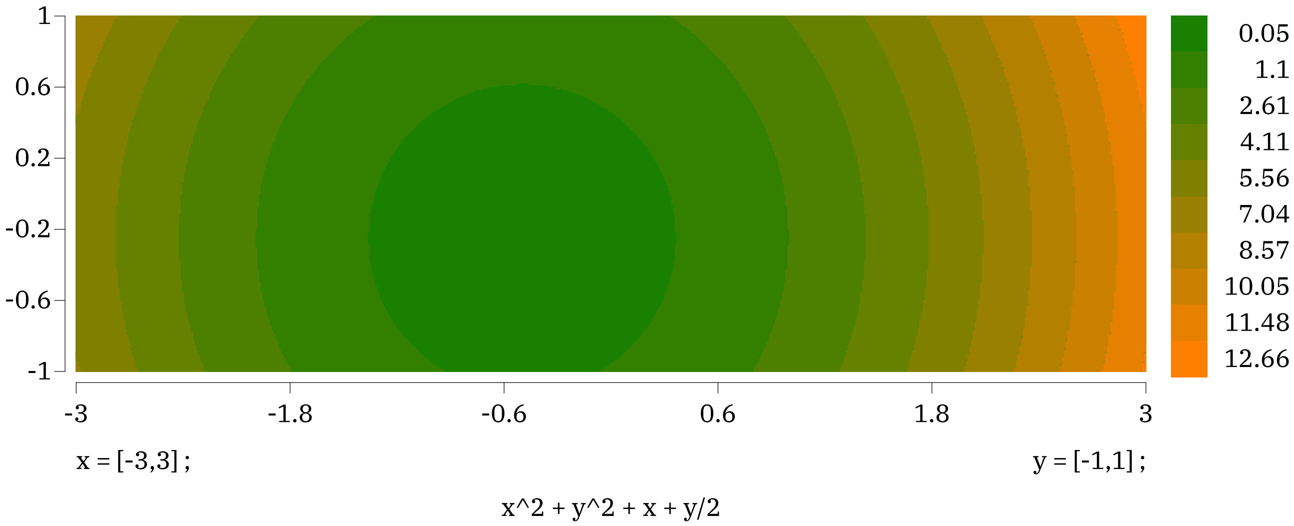


Figure 9.5

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
```

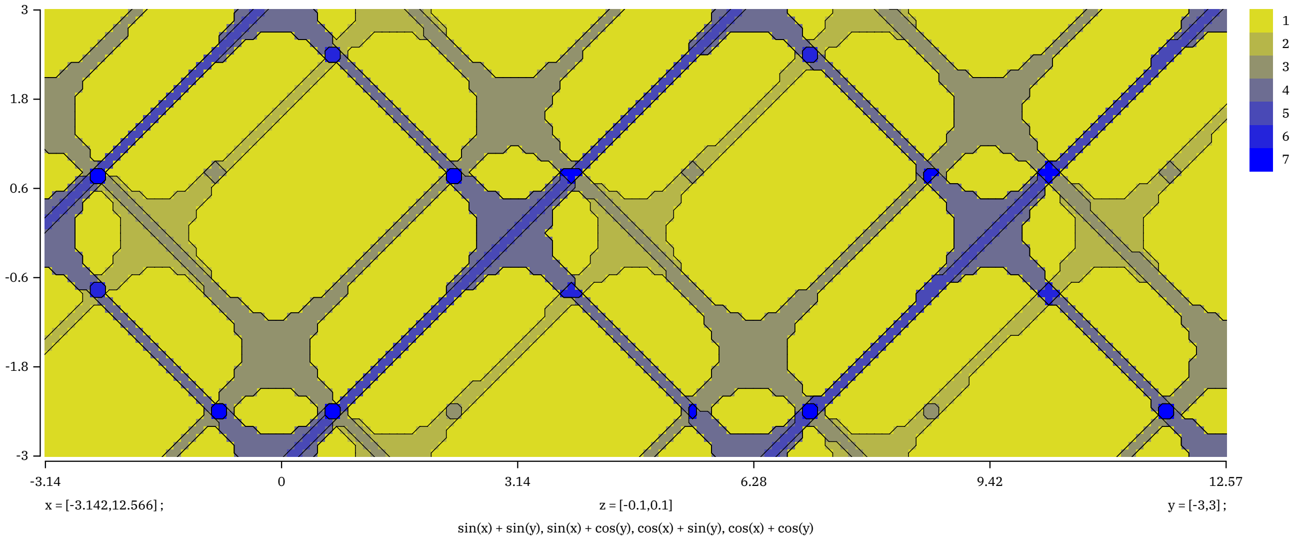


Figure 9.6

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 9.7 this is demonstrated. There we also show a variant foreground `cell`, which uses a bit different method for calculating the edges.[3]

```
\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.

---

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

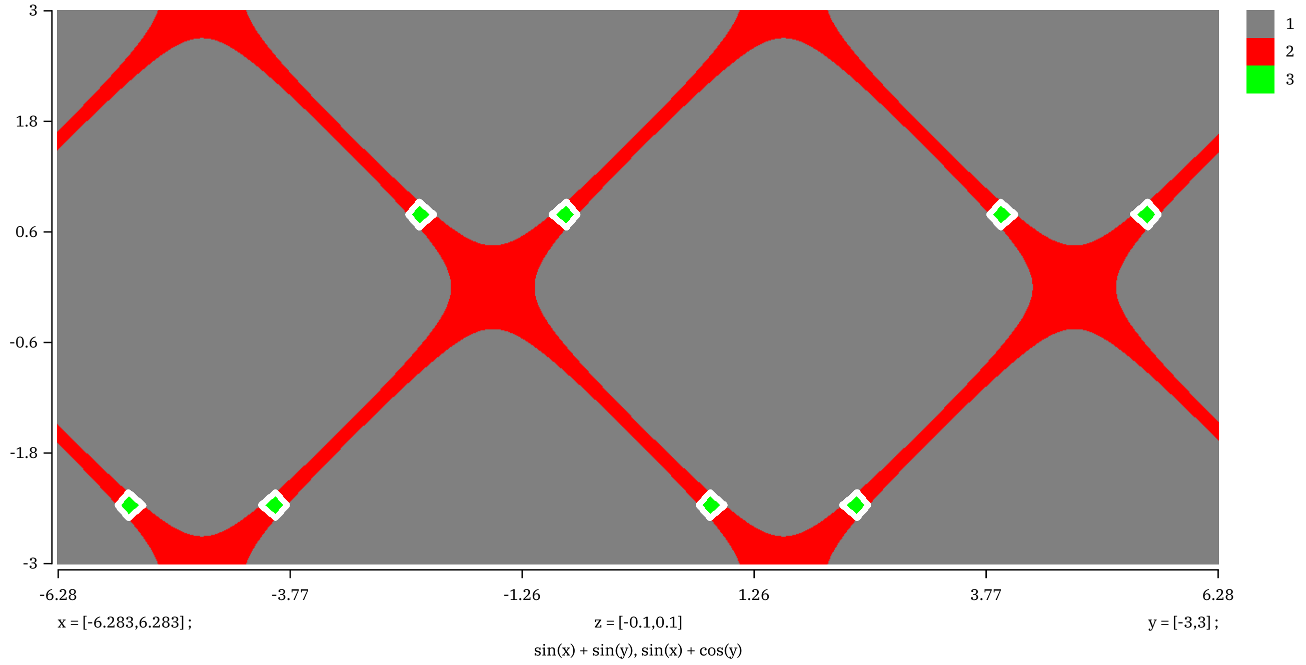


Figure 9.7

```
\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)",
            "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 9.8.

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

        range      = { -.3, .3 },
```

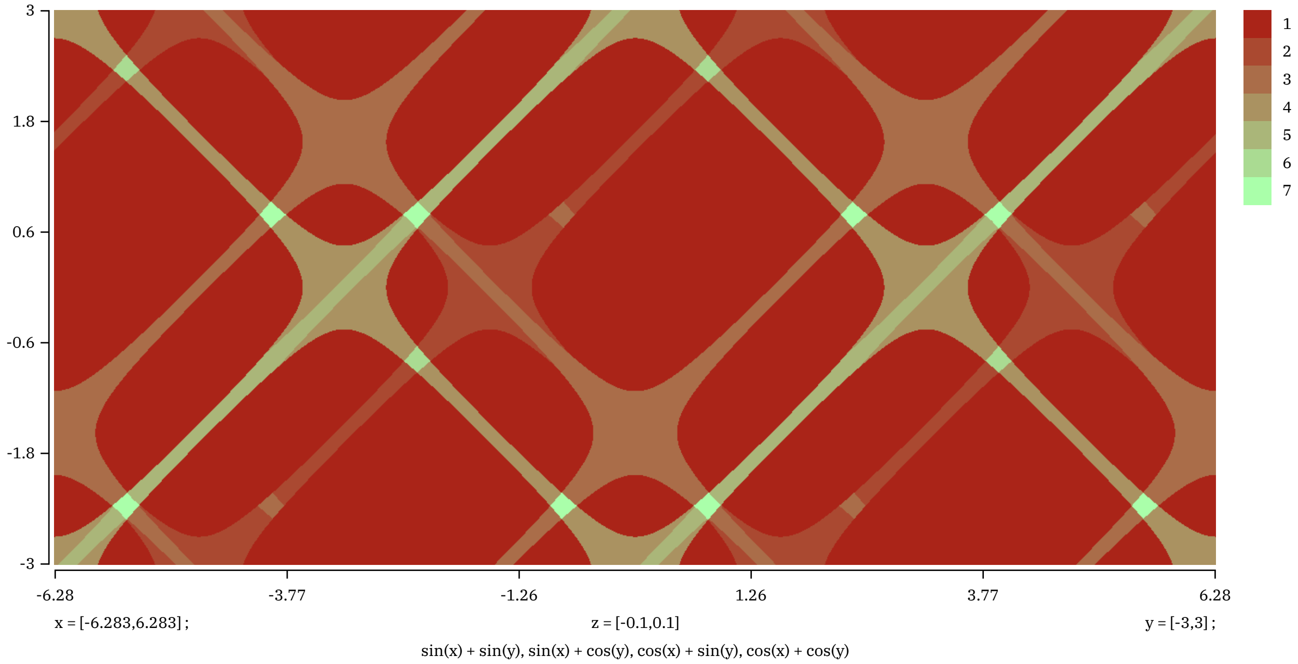


Figure 9.8

```
        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({1,0,0},{0,1,0})"
        cache      = true,
    ] xsized TextWidth ;
\stopMPcode
```

We can enlargen the window, which is demonstrated in figure 9.9. I suppose that such images only make sense in educational settings.

In figure 9.10 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})",
```

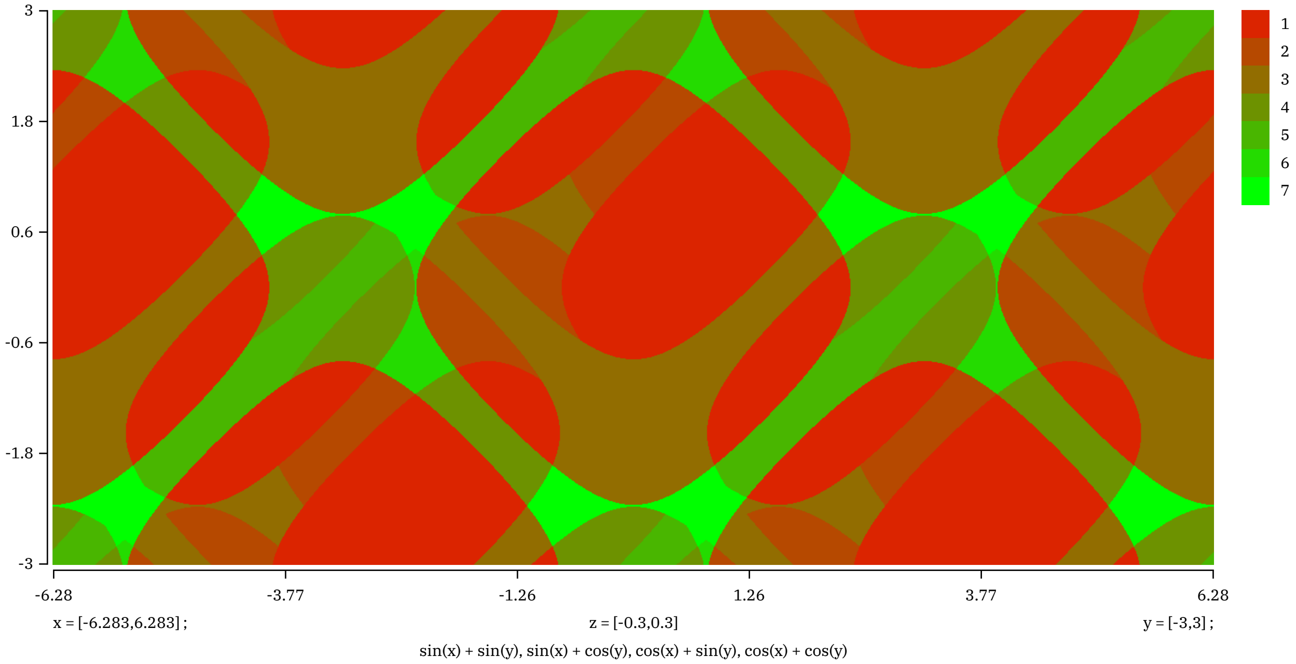


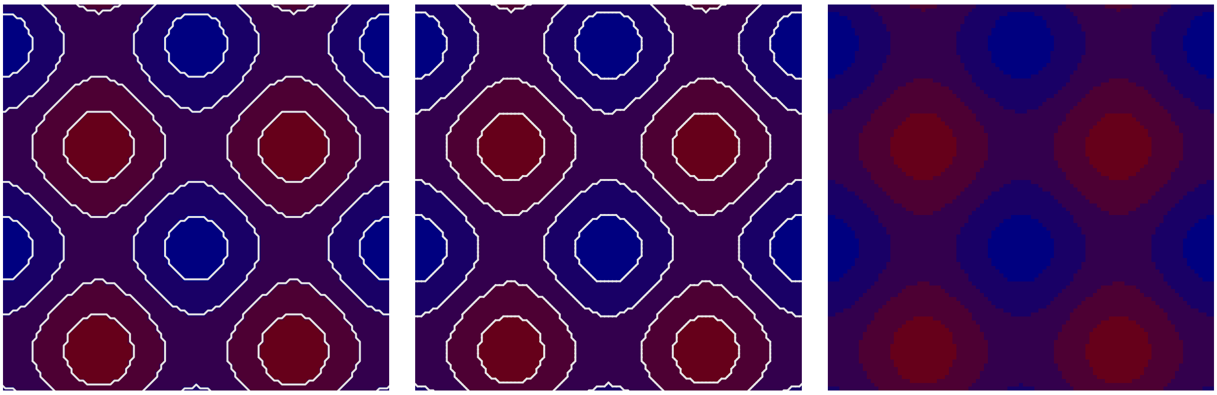
Figure 9.9

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

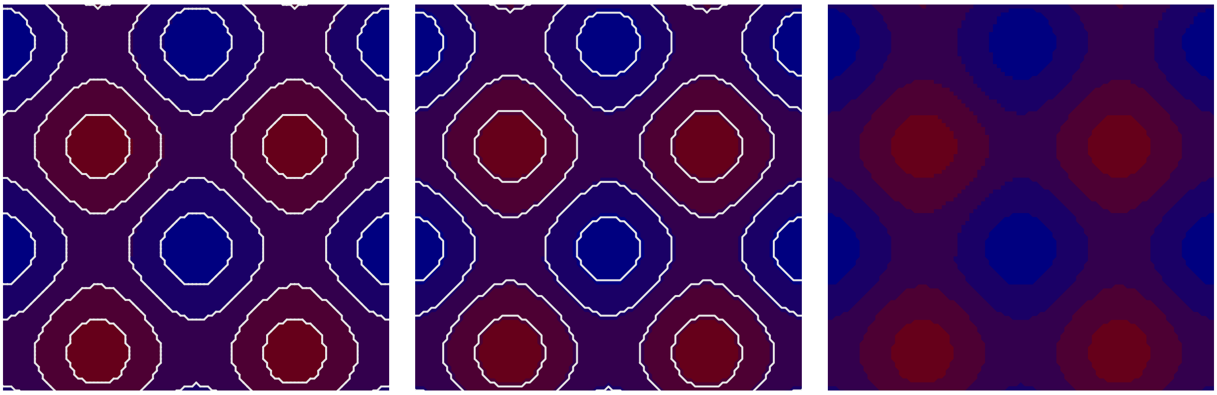
There are quite some settings. Some deal with the background, some with the foreground and quite some deal with the legend.

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`	

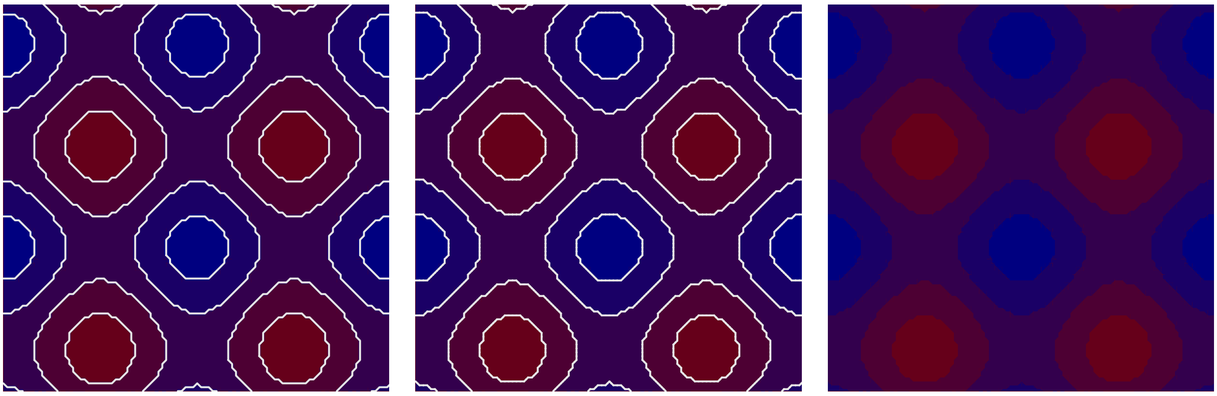
width	numeric	0	automatic when zero
height	numeric	0	automatic when zero
trace	boolean	false	
legend	string	all	x y z function range all
legendheight	numeric	LineHeight	
legendwidth	numeric	LineHeight	
legendgap	numeric	0	
legenddistance	numeric	EmWidth	
textdistance	numeric	2EmWidth/3	
functiondistance	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 9.10

10 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 10.1 we see an example of a plot with axis as well as lines drawn.

```
\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
```

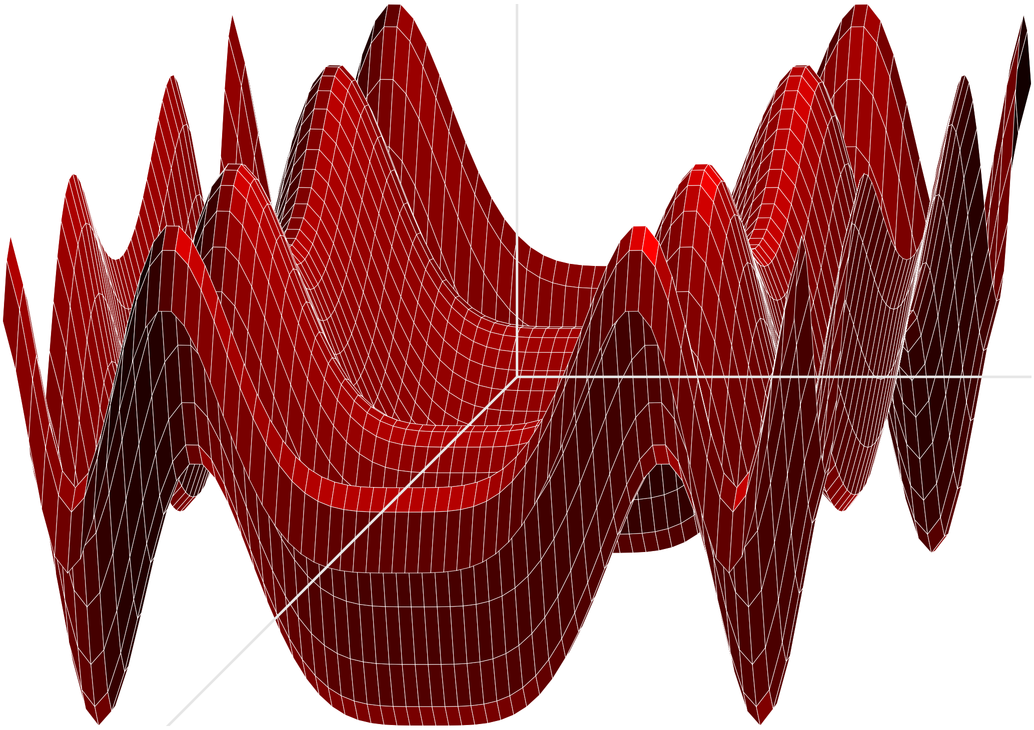


Figure 10.1

In figure 10.2 we don't draw the axis and lines. We also use a high resolution.

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

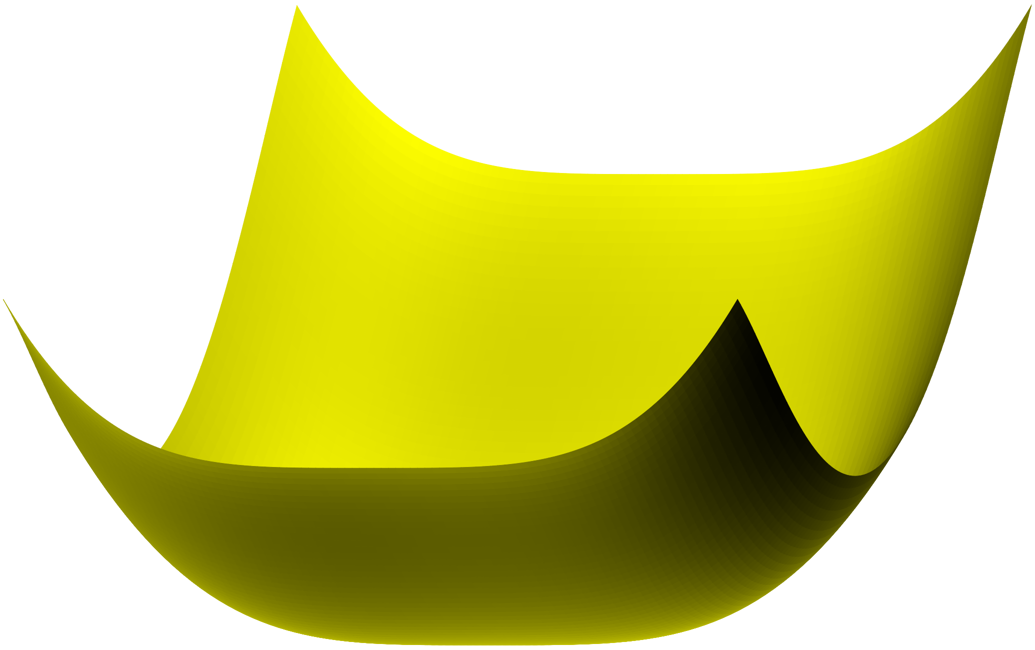


Figure 10.2

The preliminary set of parameters is:

name	**type**	**default**	**comment**	
`code`	`string`		color	string`"f, 0, 0"`
`linecolor`	`numeric`	`1`	gray scale	
`xmin`	`numeric`	`-1`		
`xmax`	`numeric`	`1`		
`ymin`	`numeric`	`-1`		
`ymax`	`numeric`	`1`		
`xstep`	`numeric`	`.1`		
`ystep`	`numeric`	`.1`		
`snap`	`numeric`	`.01`		
`xvector`	`list`	`{ -0.7, -0.7 }`		
`yvector`	`list`	`{ 1, 0 }`		
`zvector`	`list`	`{ 0, 1 }`		
`light`	`list`	`{ 3, 3, 10 }`		
`bright`	`numeric`	`100`		
`clip`	`boolean`	`false`		

`lines`	`boolean`	`true`
`axis`	`list`	`{ }`
`clipaxis`	`boolean`	`false`
`axiscolor`	`string`	`"gray"`
`axislinewidth`	`numeric`	`1/2`

11 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 11.1.

```
\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) ;
```

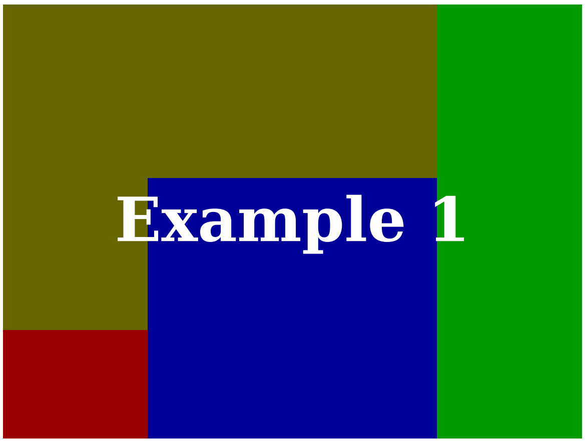
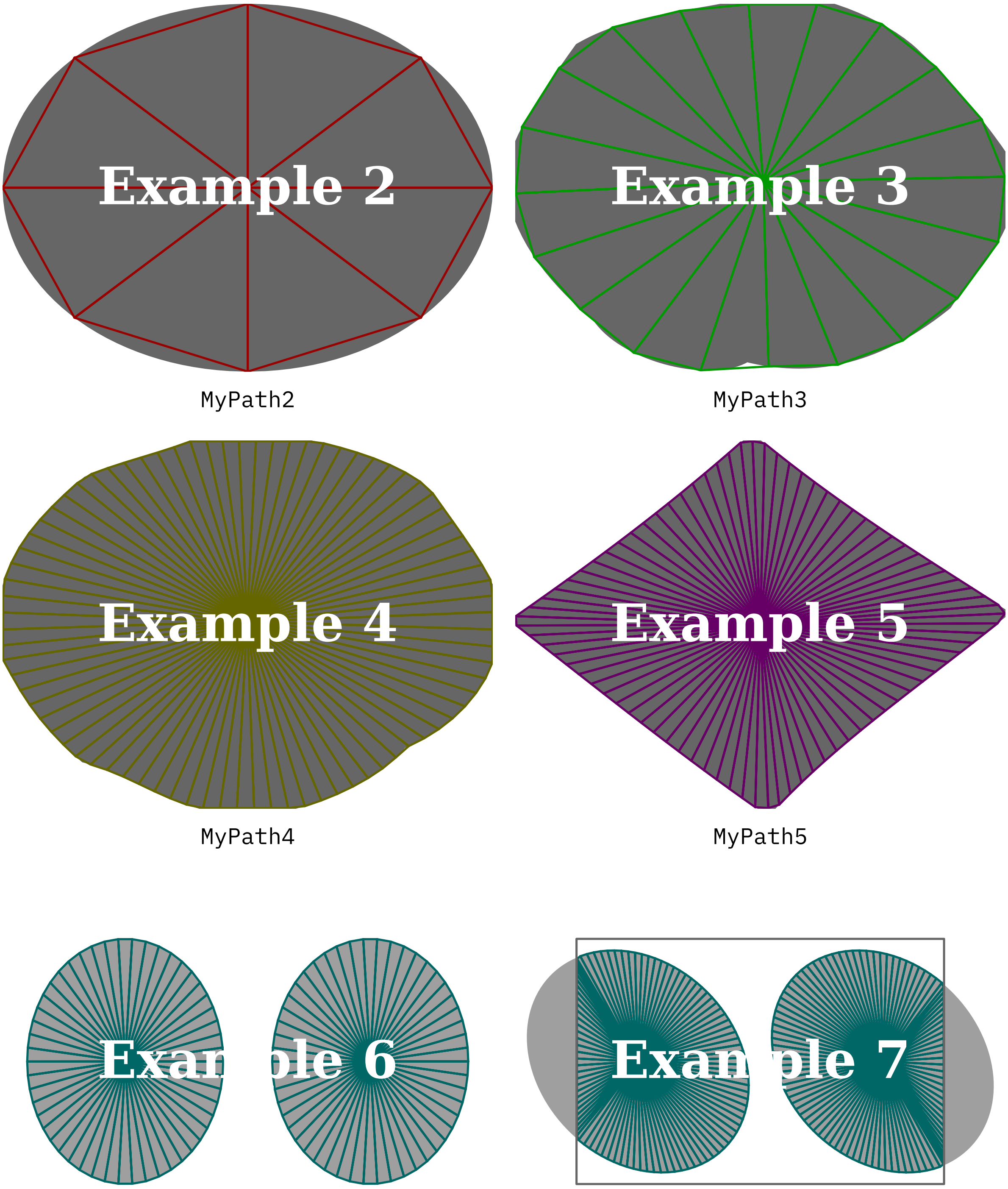


Figure 11.1

```
    save p ; path p ; p := for i=1 upto length(q) :
        (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 11.2.



MyPath2

MyPath3

MyPath4

MyPath5

MyPath6

MyPath7

Figure 11.2

12 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:

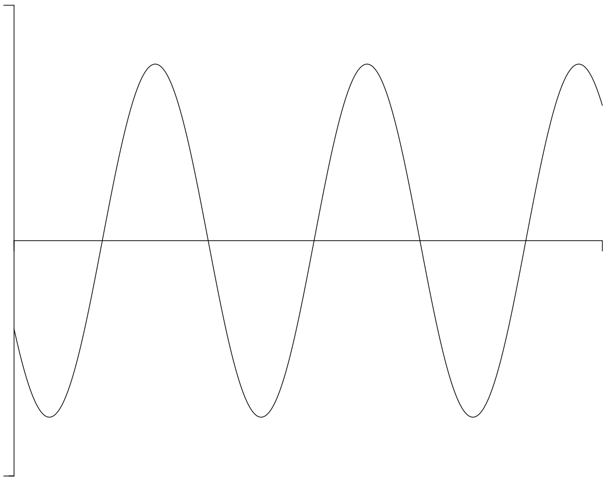


Figure 12.1

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 12.2 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"
```

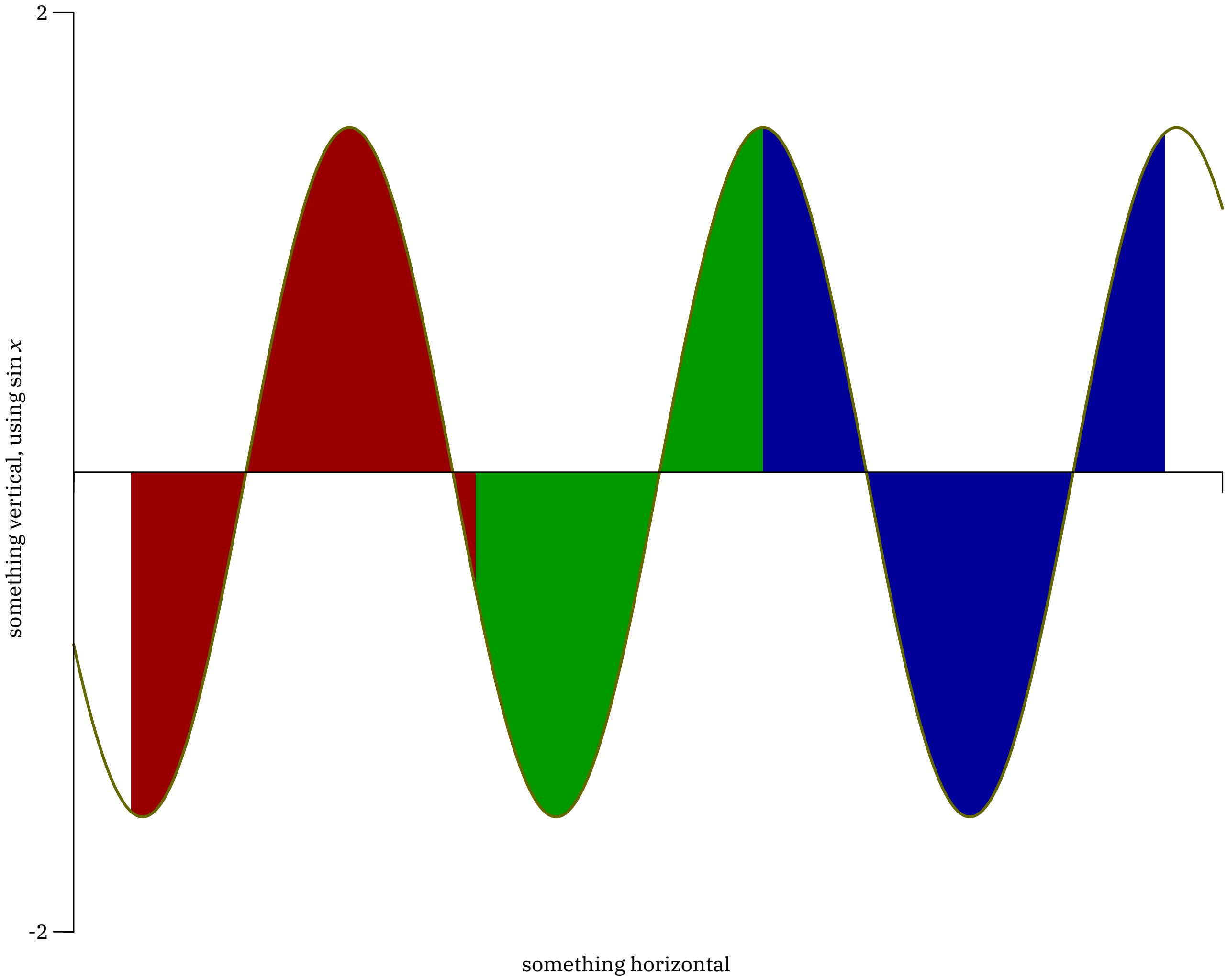


Figure 12.2

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

It is important to choose a good step. In figure 12.4 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,
```

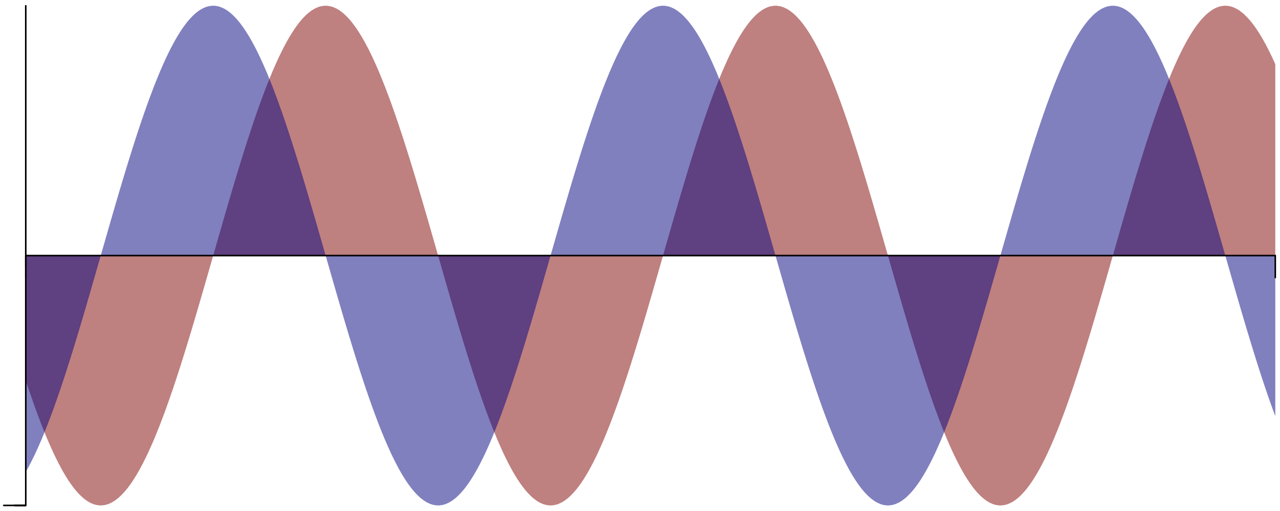


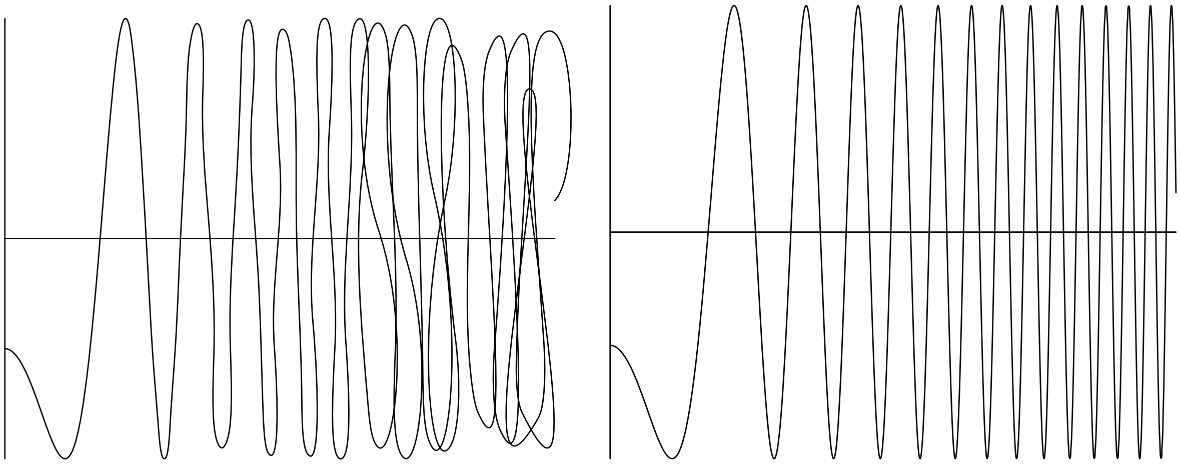
Figure 12.3

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

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

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

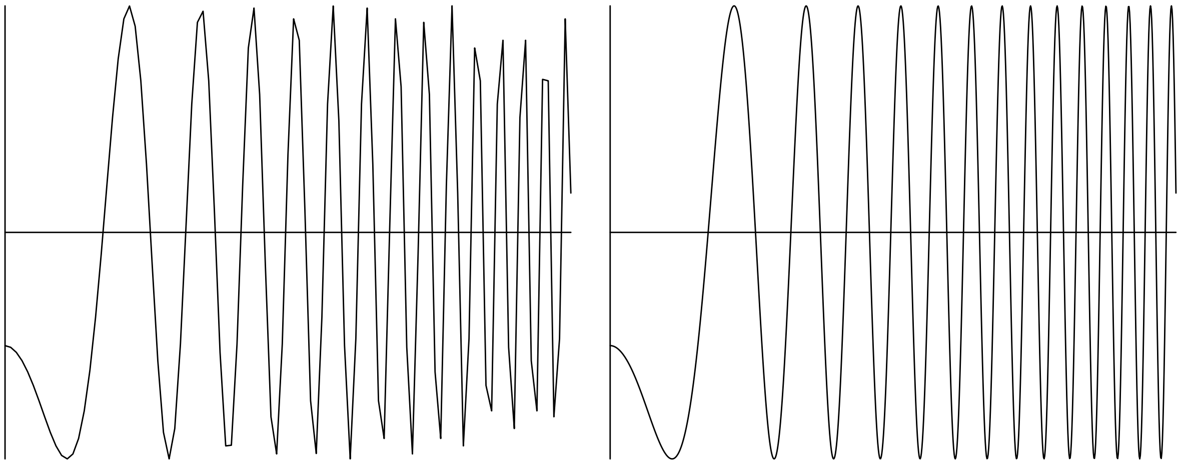
You can manipulate the axis (a bit) by tweaking the first and last ticks. In the case of figure 12.5 we also put the shape on top of the axis.

```
\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:



`xstep=.10` and `shape="curve"` `xstep=.01` and `shape="curve"`



`xstep=.10` and `shape="line"` `xstep=.01` and `shape="line"`

Figure 12.4

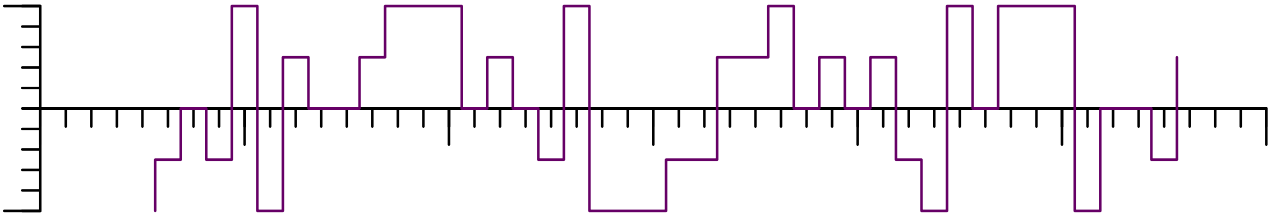


Figure 12.5

name	**type**	**default**	**comment**
`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

13 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 13.1).

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

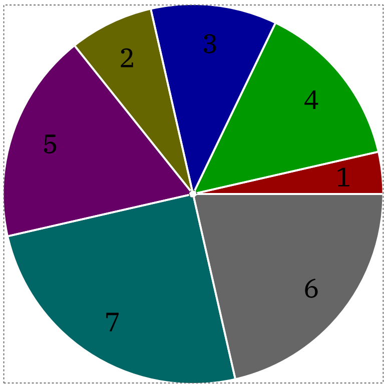


Figure 13.1

As with all these LMTX extensions, you're invited to play with the parameters. in figure 13.2 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

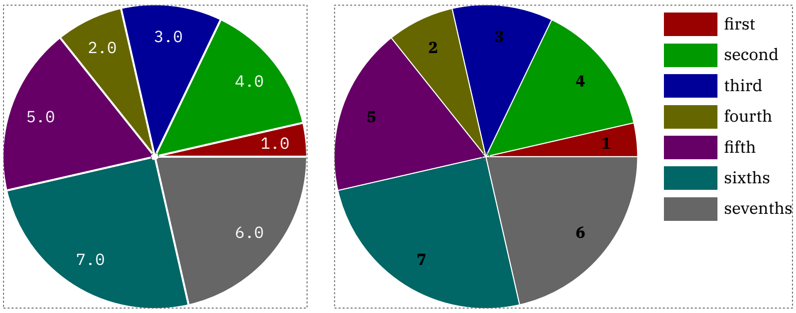


Figure 13.2

A second way of rendering are histograms, and the interface is mostly the same. In figure 13.3 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
```

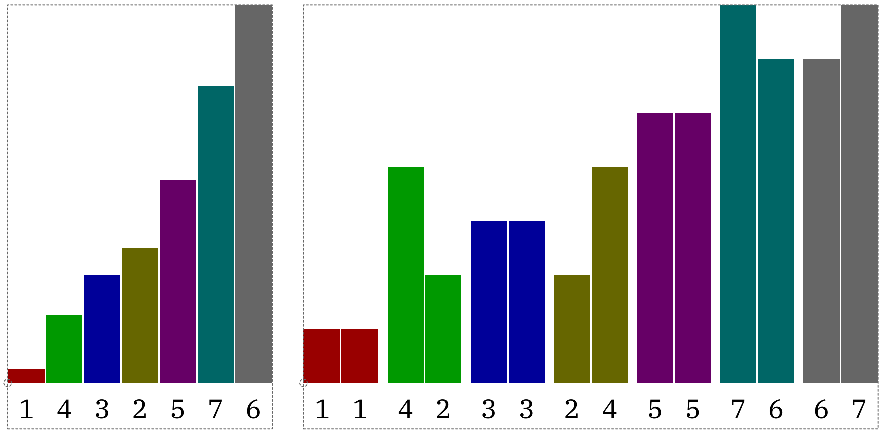


Figure 13.3

A cumulative variant is shown in figure 13.4 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
```

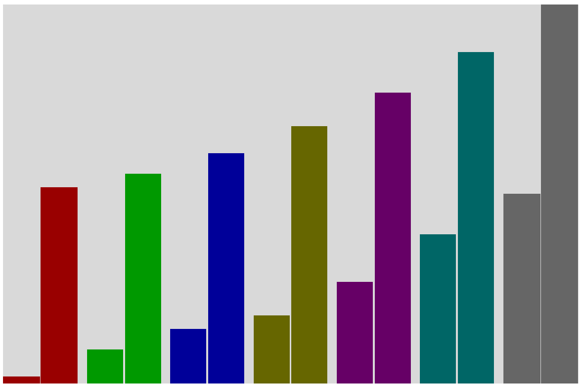


Figure 13.4

A different way of using colors is shown in figure 13.5 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
```

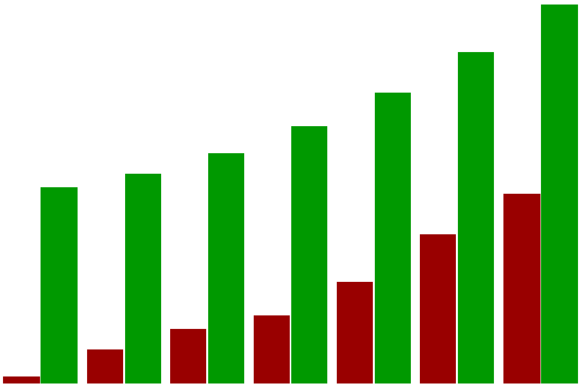


Figure 13.5

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 13.6. 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
```

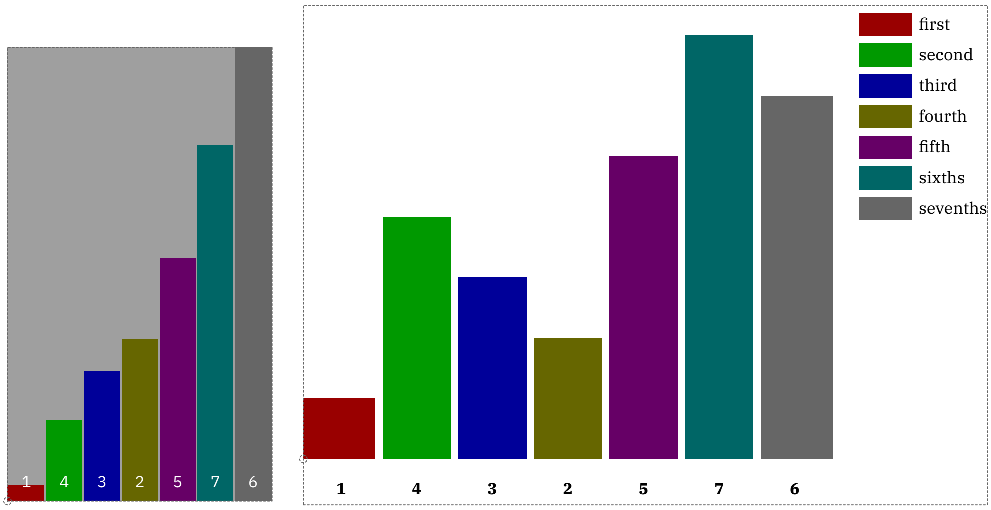


Figure 13.6

The third category concerns bar charts that run horizontal. Again we see similar options driving the rendering (figure 13.7).

```
\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
```

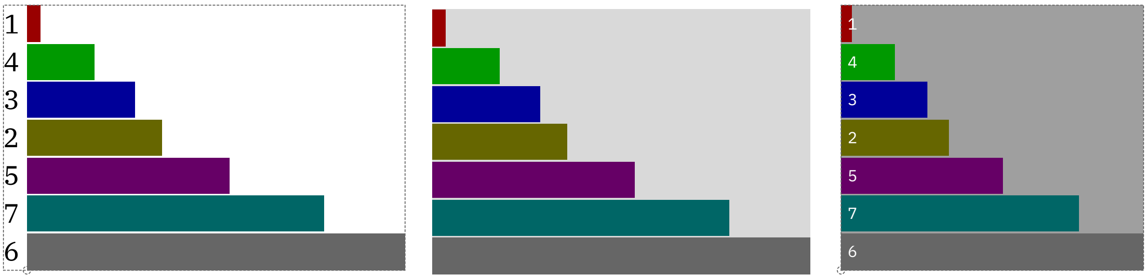


Figure 13.7

Here is one with a legend (rendered in figure 13.8):

```
\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
```

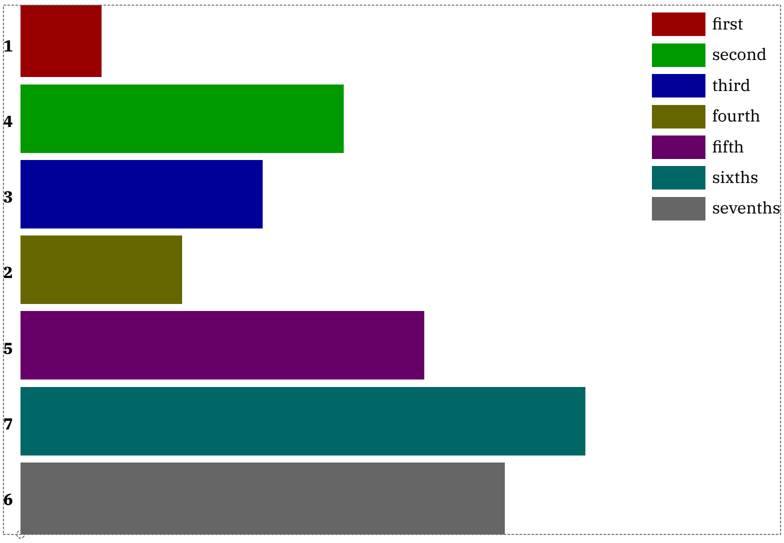


Figure 13.8

You can have labels per dataset as well as draw multiple datasets in one image, see figure 13.9:

```
\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
```

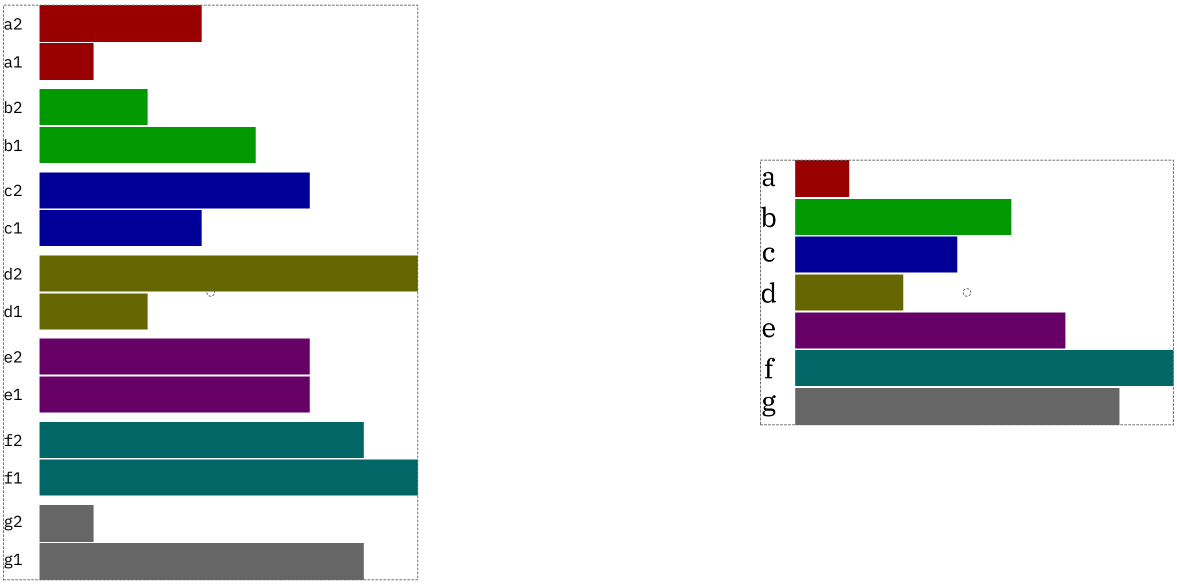


Figure 13.9

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`

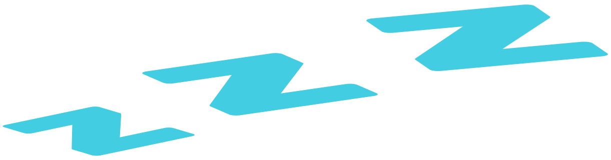
Bar charts use:

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

14 SVG

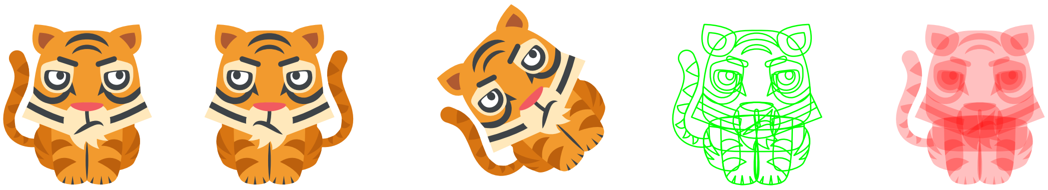
There is not that much to tell about this command. It translates an svg image to MetaPost operators. We took a few images from a mozilla emoji font:

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



Because we get pictures, you can 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		

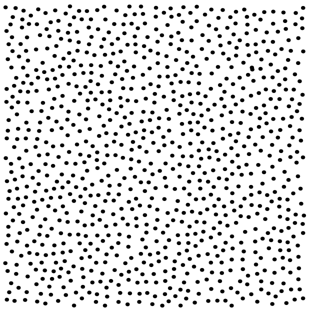
15 Poisson

When, after a post on the ConTEXt mailing list, Aditya pointed me to an article on mazes I ended up at poisson distributions which to me looks nicer than what I normally do, fill a grid and then randomize the resulting positions. With some hooks this can be used for interesting patterns too. The algorithm is based on the discussion at:

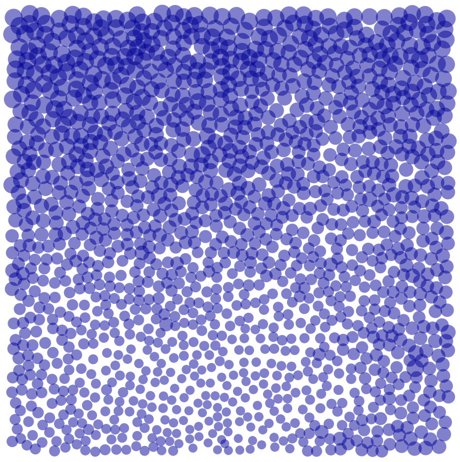
`http://devmag.org.za/2009/05/03/poisson-disk-sampling`

Other websites mention some variants on that but I saw no reason to look into those in detail. I can imagine more random related variants in this domain so consider this an appetizer. The user is rather simple because some macro is assumed to deal with the rendering of the distributed points. We just show some examples (because the interface might evolve).

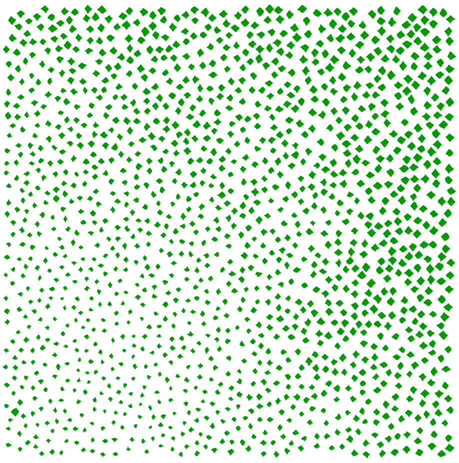
```
\startMPcode
    draw lmt_poisson [
        width    = 40,
        height   = 40,
        distance =  1,
        count    = 20,
        macro    = "draw"
    ] xsized 4cm ;
\stopMPcode
```



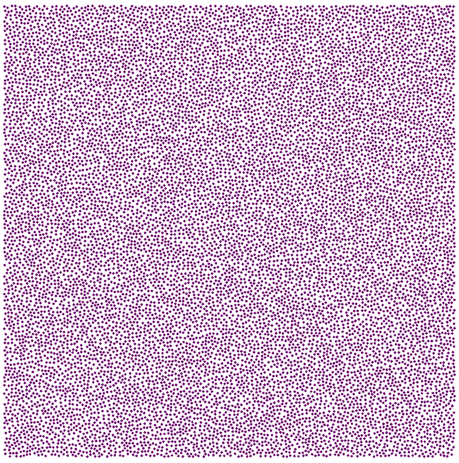
```
\startMPcode
    vardef tst (expr x, y, i, n) =
        fill fullcircle scaled (10+10*(i/n)) shifted (10x,10y)
            withcolor "darkblue" withtransparency (1,.5) ;
    enddef ;

    draw lmt_poisson [
        width     = 50,
        height    = 50,
        distance  =  1,
        count     = 20,
        macro     = "tst",
        arguments =  4
    ] xsized 6cm ;
\stopMPcode
```



```
\startMPcode
    vardef tst (expr x, y, i, n) =
        fill fulldiamond scaled (5+5*(i/n)) randomized 2 shifted (10x,10y)
            withcolor "darkgreen" ;
    enddef ;

    draw lmt_poisson [
        width     = 50,
        height    = 50,
        distance  =  1,
        count     = 20,
        macro     = "tst",
        initialx  = 10,
        initialy  = 10,
        arguments =  4
    ] xsized 6cm ;
\stopMPcode
```



```
\startMPcode{doublefun}
    vardef tst (expr x, y, i, n) =
        fill fulldiamond randomized (.2*i/n) shifted (x,y);
    enddef ;

    draw lmt_poisson [
        width     = 150,
        height    = 150,
        distance  =   1,
        count     =  20,
        macro     = "tst",
        arguments =   4
    ] xsized 6cm withcolor "darkmagenta" ;
\stopMPcode
```



```
\startMPcode
    vardef tst (expr x, y, i, n) =
        draw externalfigure "cow.pdf" ysized (10+5*i/n) shifted (10x,10y);
    enddef ;
    draw lmt_poisson [
        width     = 20,
        height    = 20,
        distance  =  1,
        count     = 20,
        macro     = "tst"
        arguments = 4,
    ] xsized 6cm ;
\stopMPcode
```

The supported parameters are:

name	**type**	**default**	**comment**
`width`	`numeric`	`50`	
`height`	`numeric`	`50`	
`distance`	`numeric`	`1`	
`count`	`numeric`	`20`	
`macro`	`string`	`"draw"`	
`initialx`	`numeric`	`10`	
`initialy`	`numeric`	`10`	
`arguments`	`numeric`	`4`	

16 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 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 `simplefun` instance because that's what is used when generating the glyphs.

```
\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:

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

    lmt_registerglyph [
        category = "test",
        unicode  = 123,
        code     = "draw TestGlyphLB ;",
        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
```

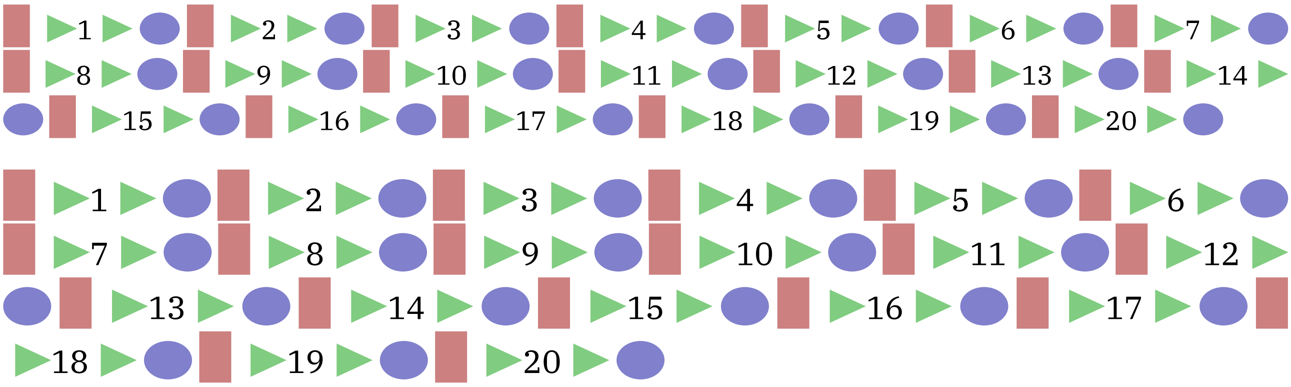
We now define a font. We always use a font as starting point which has the advantage that we always get something reasonable when we test. Of course you can use this `mps` font feature with other fonts too.

```
\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.

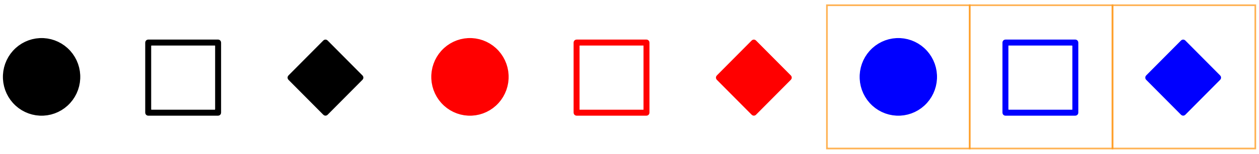
```
\startMPcalculation{simplefun}

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

    vardef SymbolBullet =
        fill unitcircle scaled  3 shifted (1.5,1.5) withpen SymbolPen
    enddef ;
    vardef SymbolSquare =
        draw unitsquare 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 Regular rendering. \stopitem
    \startitem {\MyFont\red             •  }\quad Rendering with color.
      \stopitem
    \startitem {\MyFont\blue\showglyphs •  }\quad Idem but with boundingboxes
      shown. \stopitem
\stopitemize
```

- ● □ ◆ Regular rendering.
- ● □ ◆ Rendering with color.
- ● □ ◆ Idem but with boundingboxes shown.

When blown up, these symbols look as follows:



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}

    pen KindergartenPen ; KindergartenPen := pencircle scaled 1 ;

    % 10 x 10 grid

    vardef KindergartenEqual =
        draw image
            (
                draw (2,6) -- (9,5) ;
                draw (2,4) -- (8,3) ;
            )
            shifted (0,-2)
            withpen KindergartenPen
            withcolor "KindergartenEqual"
    enddef ;
    vardef KindergartenPlus =
        draw image
            (
                draw (1,4) -- (9,5) ;
                draw (4,1) -- (5,8) ;
            )
            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"
  ] ;
  lmt_registerglyph [ category = "kindergarten", unicode = "0x002F",
    code = "KindergartenDivided"
  ] ;

\stopMPcalculation
```

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:

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

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.

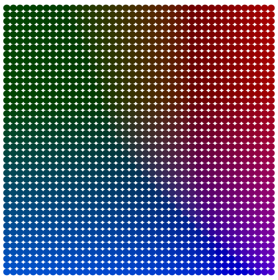
17 Color

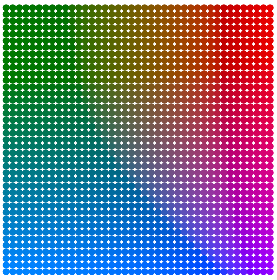
There are by now plenty of examples made by users that use color and MetaFun provides all kind of helpers. So do we need more? When I play around with things or when users come with questions that then result in a nice looking graphic, the result might en dup as example of coding. The following is an example of showing of colors. We have a helper that goes from a so called lab specification to rgb and it does that via xyz transformations. It makes no real sense to interface this beyond this converter. We use this opportunity to demonstrate how to make an interface.

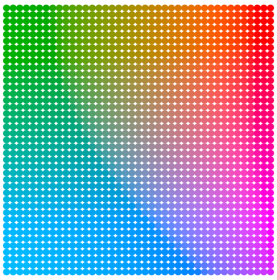
```
\startMPdefinitions
  vardef cielabmatrix(expr l, mina, maxa, minb, maxb, stp) =
    image (
      for a = mina step stp until maxa :
        for b = minb step stp until maxb :
          draw (a,b) withcolor labtorgb(l,a,b) ;
        endfor ;
      endfor ;
    )
  enddef ;
\stopMPdefinitions
```

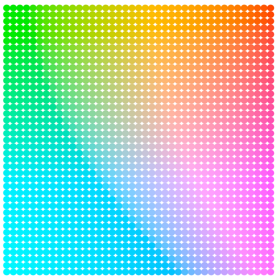
Here we define a macro that makes a color matrix. It can be used as follows

```
\startcombination[nx=4,ny=1]
  {\startMPcode draw cielabmatrix(20, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 20}}
  {\startMPcode draw cielabmatrix(40, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 40}}
  {\startMPcode draw cielabmatrix(60, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 60}}
  {\startMPcode draw cielabmatrix(80, -100, 100, -100, 100, 5) ysized 35mm
    withpen pencircle scaled 2.5 ; \stopMPcode} {\type {l = 80}}
\stopcombination
```


`l = 20`


`l = 40`

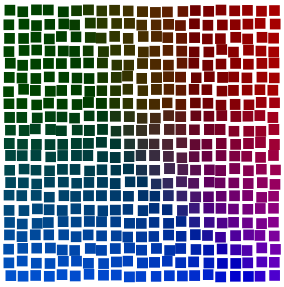

`l = 60`

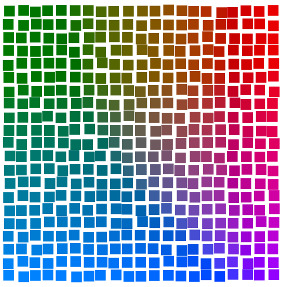

`l = 80`

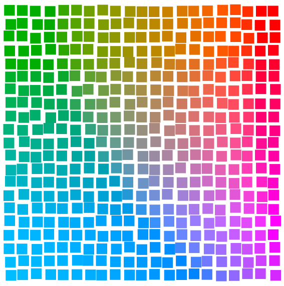
One can of course mess around a bit:

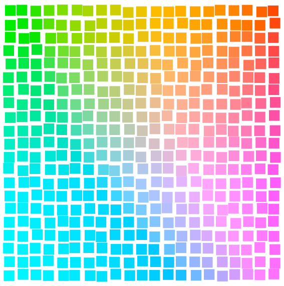
```
\startcombination[nx=4,ny=1]
  {\startMPcode draw cielabmatrix(20, -100, 100, -100, 100, 10) ysized 35mm
    randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 20}}
```

```
 {\startMPcode draw cielabmatrix(40, -100, 100, -100, 100, 10) ysized 35mm
  randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 40}}
 {\startMPcode draw cielabmatrix(60, -100, 100, -100, 100, 10) ysized 35mm
  randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 60}}
 {\startMPcode draw cielabmatrix(80, -100, 100, -100, 100, 10) ysized 35mm
  randomized 1 withpen pensquare scaled 4 ; \stopMPcode} {\type {l = 80}}
\stopcombination
```


l = 20

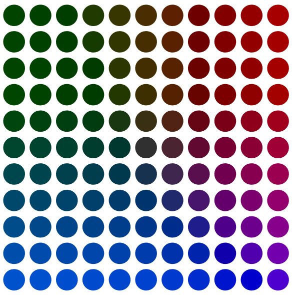

l = 40

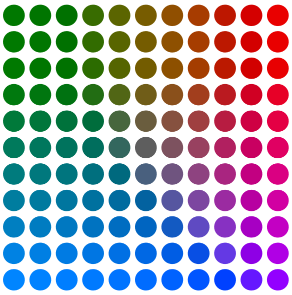

l = 60

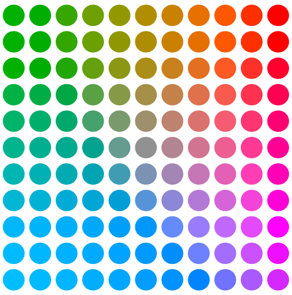

l = 80

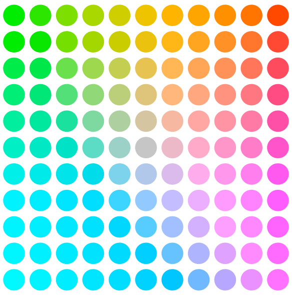
Normally, when you don't go beyond this kind of usage, a simple macro like the above will do. But when you want to make something that is upward compatible (which is one of the principles behind the ConTEXt user interface(s), you can do this:

```
\startcombination[nx=4,ny=1]
  {\startMPcode draw lmt_labtorgb [ l = 20, step = 20 ] ysized 35mm withpen
   pencircle scaled 8 ; \stopMPcode} {\type {l = 20}}
  {\startMPcode draw lmt_labtorgb [ l = 40, step = 20 ] ysized 35mm withpen
   pencircle scaled 8 ; \stopMPcode} {\type {l = 40}}
  {\startMPcode draw lmt_labtorgb [ l = 60, step = 20 ] ysized 35mm withpen
   pencircle scaled 8 ; \stopMPcode} {\type {l = 60}}
  {\startMPcode draw lmt_labtorgb [ l = 80, step = 20 ] ysized 35mm withpen
   pencircle scaled 8 ; \stopMPcode} {\type {l = 80}}
\stopcombination
```


l = 20


l = 40


l = 60


l = 80

This is a predefined macro in the reserved `lmt_` namespace (don't use that one yourself, create your own). First we preset the possible parameters:

```
presetparameters "labtorgb" [
 mina = -100,
 maxa =  100,
 minb = -100,
 maxb =  100,
 step =    5,
 l    =   50,
] ;
```

Next we define the main interface macro:

```
def lmt_labtorgb = applyparameters "labtorgb" "lmt_do_labtorgb" enddef ;
```

Last we do the actual implementation, which looks a lot like the one we started with:

```
vardef lmt_do_labtorgb =
  image (
    pushparameters "labtorgb" ;
      save l ; l := getparameter "l" ;
      for a = getparameter "mina" step getparameter "step"
            until getparameter "maxa" :
        for b = getparameter "minb" step getparameter "step"
            until getparameter "maxb" :
          draw (a,b) withcolor labtorgb(l,a,b) ;
        endfor ;
      endfor ;
    popparameters ;
  )
enddef ;
```

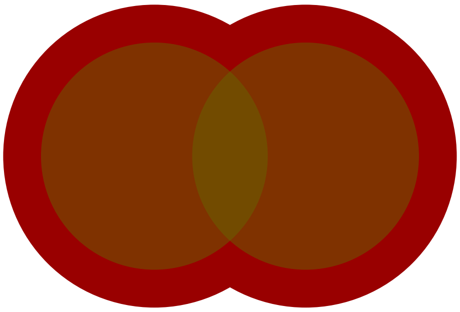
Of course we can now add all kind of extra features but this is what we currently have. Maybe this doesn't belong in the MetaFun core but it's not that much code and a nice demo. After all, there is much in there that is seldom used.

18 Groups

This is just a quick example of an experimental features.

```
\startMPcode
    fill fullcircle scaled 2cm shifted ( 5mm,2cm) withcolor "darkblue" ;
    fill fullcircle scaled 2cm shifted (15mm,2cm) withcolor "darkblue" ;

    fill fullcircle scaled 2cm shifted ( 5mm,-2cm) withcolor "darkgreen" ;
    fill fullcircle scaled 2cm shifted (15mm,-2cm) withcolor "darkgreen" ;

    draw image (
        fill fullcircle scaled 4cm                 withcolor "darkred" ;
        fill fullcircle scaled 4cm shifted (2cm,0) withcolor "darkred" ;

        setgroup currentpicture to boundingbox currentpicture
            withtransparency (1,.5) ;
    ) ;

    draw image (
        fill fullcircle scaled 3cm                 withcolor "darkyellow"
            withtransparency (1,.5) ;
        fill fullcircle scaled 3cm shifted (2cm,0) withcolor "darkyellow"
            withtransparency (1,.5) ;
    ) ;

    addbackground withcolor "darkgray" ;
\stopMPcode
```

A group create an object that when transparency is applied is treated as a group.



(Groups might become more powerful in the future, like reusable components but then some more juggling is needed.)

19 Extensions

19.1 Introduction

This is an uncorrected preliminary chapter.

The TeX and MetaPost macro languages each have their characteristics and as a result the Lua interfaces in both these subsystems are different. There are however some similarities in fetching from, scanning, and pushing back into these subsystems and by using wrappers the nasty details get hidden from users. Wrapping also permits these interfaces to evolve to a stable state.

In due time much will be documented but currently a lot is also a bit experimental because that is the way I can converge to what works best. You can assume that the solutions in the `mlib-*.lmt` files in some form stay (unless it looks too weird). Just stick to the abstractions and you will be fine.

The functionality described here is available in LMTX. Although some prototypes can be found in MkIV you should not expect the same behavior there.

19.2 The LUA interface (strings)

19.2.1 Strings

At some point the `runscript` primitive was added to mplib. Because officially the library is not bound to Lua this neutral name was chosen. In LuaMetaTeX we have a follow up on that library and although it's still neutral we just assume that Lua is used. The MetaFun follow up is therefore called LuaMetaFun, and it used the new interfaces to implement efficient going back and forth between TeX, MetaPost and Lua.

The `runscript` macro is used like this:

```
\startMPcode
draw
    textext("This will print \quotation{Hi} in the console!")
    xsized TextWidth
    withcolor "darkblue" ;
runscript("print('Hi')");
\stopMPcode
```

This will print “Hi” in the console!

The `runscript` primitive triggers a callback that gets the string passed. This callback then does some magic, normally compiling that string into byte code and execute it. The compiled function can return a string that is then fed back into the MetaPost `scantokens` primitive command. So, that return value has to be valid MetaPost!

```
\startMPcode
string s ;
s := runscript("mp.quoted('This will return a string!')") ;
draw textext(s)
    xsized TextWidth
    withcolor "darkgreen" ;
\stopMPcode
```

This will return a string!

The `mp.quoted` call is one of the build into ConTEXt ways to pipe back something to MetaPost. We will cover this later. If you don't want to use that feature, the call would have looked like this:

```
\startMPcode
string s ;
s := runscript("return " &
    "'" & ditto &
    "Ditto is a string that contains a double qoute!"
    & ditto & "'"
) ;
draw textext(s)
    xsized TextWidth
    withcolor "darkred" ;
\stopMPcode
```

Ditto is a string that contains a double qoute!

The `ditto` with ampersands trickery constructs a string with embedded quotes which is needed because you want to pass back a string and MetaPost only considers something a string when it sees double quotes.

19.2.2 Numerics

Instead of a string you can also pass a numeric:

```
\startMPcode
runscript 10000;
\stopMPcode
```

This time, on the console you will see something:

```
metapost > lua > 1: bad index: 10000
metapost > lua > 1: no result, invalid code: 10000
```

This I because at the Lua end this number should result in some action, in the case of ConTEXt calling a registered function. Because the given number is unknown nothing is done. These messages come from ConTEXt, and MetaPost will keep silent because we don't pass anything back.

This numeric interface only makes sense when the callback handles it and the way ConTEXt does that is probably unique to that macro package. You can of course create MetaPost instances yourself (in Lua) and handle callbacks your own way: you get a string, do this, you get a number, do that.

19.2.3 Helpers

In order to help users passing data to the Lua end there are some helper macros defined using the `lua` macro with suffixes:

```
draw lua.mp.foo(0,2,(3,4)) ;
fill lua.MP.foo(0,2,(3,4)) ;
```

The lowercase `mp` namespace is for ConTEXt itself so if you use that for your own extensions, there is no guarantee against future clashes. The uppercase `MP` namespace is for users. In any case you need to be aware of expansion, so `foo` should not expand to something weird (variable names and `vardef` macro names are okay).

At the Lua end these are mapped onto functions, like:

```
function mp.foo(n,m,p)
    -- do something
end
function MP.foo(n,m,p)
    -- do something
end
```

19.3 Printing back

In the previous chapter we saw `mp.quoted` being used to print back a string to MetaPost for processing by `scantokens`. Not all function in the `mp` namespace are meant for usage, so best stick to what is described here.

The most generic print is `mp.print` that takes multiple arguments. A numeric value is flushed as serialized number and a string is passed along (so no quotes are added). A boolean becomes `true` or `false`. A table with six elements is seen as a transform and otherwise passed as pair, color or cmyk color definition. The `print` command takes multiple arguments and the results are concatenated into one string with other prints so far.

Because this mechanism is already available in MkIV we remain compatible which means that the print functions are available in the `mp` namespace but also in the `mp.aux` namespace. In the meantime we moved to the `print` namespace. The main print command does a guess about what it is fed and will inject that as string. Thereby the next are all valid:

```
fill fullcircle scaled runscript("mp.print      ('3cm')") withcolor "darkred" ;
fill fullcircle scaled runscript("mp.print.print('2cm')") withcolor "darkgreen" ;
fill fullcircle scaled runscript("mp.aux.print  ('1cm')") withcolor "darkblue" ;
```

`string`	`string`	passed as it is but with percent, double quote and newline escaped
`boolean`	`boolean`	the true or false primitives
`integer`	`number`	an integer
`number`	`number`	a float
`numeric`	`number`	a float (same as previous)
`points`	`number`	a scaled numeric with `pt` unit
`pair`	`numbers or table`	a pair (x,y) or (x,x)
`pairpoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`triplet`	`numbers or table`	a rgb triplet (r,g,b)
`tripletpoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`quadruple`	`numbers or table`	a cmyk quadruple (c,m,y,k)
`quadruplepoints`	`numbers or table`	idem but with scaled numbers and a `pt` unit
`color`	`numbers or table`	a numeric, triplet or quadruple
`transform`	`numbers or table`	a six element transform
`print`	`whatever`	the normal semi-intelligent printer
`fprint`	`format, whatever`	the normal semi-intelligent printer using a format
`vprint`	`variable`	the normal semi-intelligent printer with escaped percents, quotes and newlines
`quoted`	`string`	a valid string surrounded by quotes with an optional first format specifier

A more complex printer is `path` that takes upto three arguments. The first argument is a table. Entries have two or six elements where the last two are control points. The second argument indicates the connector: `true` and `nil` indicate `..` while `false` will use `--`. When the last argument is true we have a closed path. Alternatively the table can have a boolean `cycle` field. So these are all valid:

```
local t1 = { {0,0}, {1,0}, {1,1}, {0,1} }
local t2 = { {0,0}, {1,0}, {1,1}, {0,1}, cycle = true }

mp.print.path(t1)
mp.print.path(t1,nil,true)
mp.print.path(t1,true,true)
mp.print.path(t1,false)
mp.print.path(t1,false,true)
mp.print.path(ts,false)
mp.print.path(t1,"...",true)
mp.print.path(t1,"..",true)
mp.print.path(t2,"..")
```

As with the already mentioned simple printers there is a variant that scales: `pathpoints` (an alternative is of course to scale the whole path by `pt`).

The result of what goes into the print functions is collected and flushed to MetaPost at the end of a call. You can directly push something in the buffer with `mp.direct` and condense the (so far) buffered content with `mp.flush`. Normally you will not need such low level handling.

19.4 Direct values

The print functions accumulate and flush at the end. Alternatively you can return a value. In that case the type determines what gets done:

`number`	native quantity
`boolean`	native quantity (*I need to check this!*)
`string`	feeds into `scantokens`
`table`	feeds concatenated into `scantokens`

Instead of return you can also call an injector. The repertoire is similar to the printers: `boolean`, `cmykcolor`, `color`, `integer`, `number`, `numeric`, `pair`, `path`, `quadruplet`, `string`, `transform`, `triplet` and `whatever` (kind of automatic):

```
function MP.MyFunction()
    mp.inject.string("This is just a string.")
end
```

The `whd`, `xy` and `pt` injectors inject triplets, pairs and numeric scaled from TEX scaled points to base points.

19.5 Registering

Quite some of the build in functionality uses a slightly different approach. It roughly works as follows:

```
% reserve an index and set its value:

newscriptindex user_me_foo ; user_me_foo := scriptindex "user_me_foo" ;

% wrap the call into a macro:

def me_foo = runscript user_me_foo enddef ;
```

A macro can of course be more complex, for instance take arguments and push those into the script call:

```
def me_foo(expr a, b) = runscript user_me_foo a b enddef ;
```

But before this is done at the MetaPost end, you need to define the Lua function:

```
local function user_me_foo()
    -- do something useful
end

metapost.registerscript("user_me_foo",user_me_foo)
```

In this case you use the print and inject functions, of course only when you want to push back some result.

Alternatively you can do:

```
metapost.registerdirect("user_me_foo",user_me_foo)
metapost.registertokens("user_me_foo",user_me_foo)
```

A direct script will treat return values as native, so string and tables are like quoted string and interpreted objects (boolean, numeric, tables). The tokens variant will feed the strings and concatenated tables into scantokens.

The script index can be fetched at the Lua end with:

```
local index = metapost.scriptindex(name)
```

19.6 Codes and such

Using the to be discussed scanners assumes that you know some of the internals (or at least concepts) of MetaPost. Taco has written some excellent tutorials on the way MetaPost handles input. Here we just mention what you can run into.

Each primitive, macro or variable falls into a category. The primitives are grouped in a way that permits handling them as category and the following table shows the grouping. Internally the subcategories are called modes. You should treat these numbers as abstractions because they can change over time, depending on how the library evolves. Modes can normally be ignored.

code	mode	name	code category
64	1	#@	macrospecial
51	100	&	ampersand
58	88	*	secondarybinary
46	86	+	plusorminus
48	90	++	tertiarybinary
48	91	+-+	tertiarybinary
78	0	,	comma
46	87	-	plusorminus
50	0	..	pathjoin
57	89	/	slash
77	0	:	colon
76	0	:=	assignment
79	0	;	semicolon
53	94	<	primarybinary
53	95	<=	primarybinary
53	99	<>	primarybinary
54	98	=	equals
53	96	>	primarybinary
53	97	>=	primarybinary
64	2	@	macrospecial
64	3	@#	macrospecial
36	56	ASCII	unary
66	0	[	leftbracket
9	0	\	relax
67	0	]	rightbracket
21	0	addto	addto

70	2	also	thingstoadd
55	93	and	and
36	79	angle	unary
36	78	arclength	unary
40	118	arctime	ofbinary
62	0	atleast	atleast
26	1	batchmode	mode
34	0	begingroup	begingroup
36	17	blackpart	unary
36	13	bluepart	unary
32	21	boolean	typename
36	85	bounded	unary
40	121	boundingpath	ofbinary
1	0	btex	btex
36	57	char	unary
43	21	charcode	internal
43	24	chardp	internal
43	23	charht	internal
43	25	charic	internal
43	22	charwd	internal
22	36	clip	setbounds
36	83	clipped	unary
36	45	closefrom	unary
32	28	cmykcolor	typename
32	27	color	typename
36	60	colormodel	unary
70	1	contour	thingstoadd
60	0	controls	controls
36	71	cosd	unary
63	0	curl	curl
36	14	cyanpart	unary
39	80	cycle	cycle
69	1	dashed	with
36	63	dashpart	unary
19	1	def	macrodef
30	0	delimiters	delimiters
40	113	directiontime	ofbinary
70	0	doublepath	thingstoadd
4	3	else	fiorelse
4	4	elseif	fiorelse
19	0	enddef	macrodef
6	0	endfor	iteration
80	0	endgroup	endgroup
5	1	endinput	input
40	120	envelope	ofbinary
28	2	errhelp	message
28	1	errmessage	message
26	4	errorstopmode	mode

2	0	etex	etex
29	0	everyjob	everyjob
8	0	exitif	exittest
13	0	expandafter	expandafter
59	8	expr	parametertype
35	38	false	nullary
4	2	fi	fiorelse
36	81	filled	unary
36	72	floor	unary
6	2	for	iteration
6	1	forever	iteration
6	3	forsuffixes	iteration
36	12	greenpart	unary
36	18	greypart	unary
36	84	grouped	unary
36	55	hex	unary
3	1	if	if
56	0	infont	primarydef
23	0	inner	protection
5	0	input	input
16	0	interim	interim
48	109	intersectiontimes	tertiarybinary
43	3	jobname	internal
36	47	known	unary
36	58	length	unary
17	0	let	let
43	31	linecap	internal
43	30	linejoin	internal
36	74	llcorner	unary
36	75	lrcorner	unary
36	15	magentapart	unary
36	52	makepath	unary
36	53	makepen	unary
12	0	maketext	maketext
36	68	mexp	unary
43	33	miterlimit	internal
36	69	mlog	unary
35	119	mpversion	nullary
18	0	newinternal	newinternal
26	2	nonstopmode	mode
35	43	normaldeviate	nullary
36	49	not	unary
35	40	nullpen	nullary
35	39	nullpicture	nullary
43	2	numberprecision	internal
43	1	numbersystem	internal
32	30	numeric	typename
36	54	oct	unary

36	46	odd	unary
71	0	of	of
48	92	or	tertiarybinary
23	1	outer	protection
43	29	overloadmode	internal
32	29	pair	typename
32	24	path	typename
36	61	pathpart	unary
43	26	pausing	internal
32	23	pen	typename
35	42	pencircle	nullary
40	117	penoffset	ofbinary
36	62	penpart	unary
32	25	picture	typename
40	114	point	ofbinary
40	116	postcontrol	ofbinary
36	65	postscriptpart	unary
40	115	precontrol	ofbinary
36	64	prescriptpart	unary
59	1	primary	parametertype
19	3	primarydef	macrodef
27	0	randomseed	randomseed
36	44	readfrom	unary
35	41	readstring	nullary
36	11	redpart	unary
43	37	restoreclipcolor	internal
36	51	reverse	unary
32	27	rgbcolor	typename
58	101	rotated	secondarybinary
11	0	runscript	runscript
15	0	save	save
58	103	scaled	secondarybinary
10	0	scantokens	scantokens
26	3	scrollmode	mode
59	2	secondary	parametertype
19	4	secondarydef	macrodef
22	38	setbounds	setbounds
22	37	setgroup	setbounds
24	1	setproperty	property
58	104	shifted	secondarybinary
20	0	shipout	shipout
25	2	show	show
25	4	showdependencies	show
25	1	showstats	show
43	27	showstopping	internal
25	0	showtoken	show
25	3	showvariable	show
26	5	silentmode	mode

36	70	sind	unary
58	102	slanted	secondarybinary
36	67	sqrt	unary
43	32	stacking	internal
36	66	stackingpart	unary
73	0	step	step
37	0	str	str
32	22	string	typename
36	82	stroked	unary
40	112	subpath	ofbinary
40	111	substring	ofbinary
59	9	suffix	parametertype
61	0	tension	tension
59	3	tertiary	parametertype
19	5	tertiarydef	macrodef
43	28	texscriptmode	internal
59	10	text	parametertype
43	18	time	internal
72	0	to	to
43	6	tracingcapsules	internal
43	7	tracingchoices	internal
43	9	tracingcommands	internal
43	5	tracingequations	internal
43	11	tracingmacros	internal
43	14	tracingonline	internal
43	12	tracingoutput	internal
43	10	tracingrestores	internal
43	8	tracingspecs	internal
43	13	tracingstats	internal
43	4	tracingtitles	internal
32	26	transform	typename
58	105	transformed	secondarybinary
35	37	true	nullary
43	35	truecorners	internal
36	59	turningnumber	unary
36	76	ulcorner	unary
36	73	uniformdeviate	unary
36	48	unknown	unary
74	0	until	until
36	77	urcorner	unary
19	2	vardef	macrodef
1	1	verbatimtex	btex
38	0	void	void
43	34	warningcheck	internal
69	9	withcmykcolor	with
69	6	withgreyscale	with
75	0	within	within
69	5	withoutcolor	with

69	0	withpen	with
69	3	withpostscript	with
69	2	withprescript	with
69	8	withrgbcolor	with
69	4	withstacking	with
31	0	write	write
36	5	xpart	unary
58	106	xscaled	secondarybinary
36	7	xxpart	unary
36	8	xypart	unary
36	16	yellowpart	unary
36	6	ypart	unary
58	107	yscaled	secondarybinary
36	9	yxpart	unary
36	10	yypart	unary
58	108	zscaled	secondarybinary
49	0	{	leftbrace
68	0	}	rightbrace

Variables are of a certain type. Possible variable types are available in `metapost.types` via numeric and verbose keys: 0: undefined, 1: vacuous, 2: boolean, 3: unknownboolean, 4: string, 5: unknownstring, 6: pen, 7: unknownpen, 8: path, 9: unknownpath, 10: picture, 11: unknownpicture, 12: transform, 13: color, 14: cmykcolor, 15: pair, 16: numeric, 17: known, 18: dependent, 19: protodependent, 20: independent, 21: tokenlist, 22: structured, 23: unsuffixedmacro, 24: suffixedmacro.

The possible command codes (as seen in the primitive table) are available in `metapost.codes` via numeric and verbose keys: 0: undefined, 1: btex, 2: etex, 3: if, 4: fiorelse, 5: input, 6: iteration, 7: repeatloop, 8: exittest, 9: relax, 10: scantokens, 11: runscript, 12: maketext, 13: expandafter, 14: definedmacro, 15: save, 16: interim, 17: let, 18: newinternal, 19: macrodef, 20: shipout, 21: addto, 22: setbounds, 23: protection, 24: property, 25: show, 26: mode, 27: randomseed, 28: message, 29: everyjob, 30: delimiters, 31: write, 32: typename, 33: leftdelimiter, 34: begingroup, 35: nullary, 36: unary, 37: str, 38: void, 39: cycle, 40: ofbinary, 41: capsule, 42: string, 43: internal, 44: tag, 45: numeric, 46: plusorminus, 47: secondarydef, 48: tertiarybinary, 49: leftbrace, 50: pathjoin, 51: ampersand, 52: tertiarydef, 53: primarybinary, 54: equals, 55: and, 56: primarydef, 57: slash, 58: secondarybinary, 59: parametertype, 60: controls, 61: tension, 62: atleast, 63: curl, 64: macrospecial, 65: rightdelimiter, 66: leftbracket, 67: rightbracket, 68: rightbrace, 69: with, 70: thingstoadd, 71: of, 72: to, 73: step, 74: until, 75: within, 76: assignment, 77: colon, 78: comma, 79: semicolon, 80: endgroup, 81: stop, 82: undefinedcs.

When you scan for input not all of these make sense, often you will stick to dealing with symbols like brackets, braces, equal signs and variables or expressions.

19.7 Scanners

The most low level scanners are `token` and `symbol`. Although we have them in the `mp.scan` namespace they are just library calls. You use them like:

```
if scan.symbol(true) == "[" then -- "]"
    scan.symbol()
```

```
else
    ...
end
```

Here we check if the upcoming token is a specific symbol. The `true` will push back the token. A second boolean argument will enforce expansion.

Scanning can be hairy because the engine is set up in a way that mix lookahead, expand, resolve and processing. So, you can run into a numeric constant, but also in a not yet resolved quantity (take = versus `:=`). When writing more complex scanners it helps to print codes and types.

The `scan.token` function returns a command, mode and expression type but in practice you only have to consider the first value. Other scanners are `boolean`, `cmykcolor`, `color`, `expression`, `integer`, `next`, `number`, `numeric`, `pair`, `path`, `pen`, `property`, `string`, `transform`, plus some implemented around these. Keep in mind that scanners are bound to an instance so the functions in the `scan` namespace are actually wrappers around the library calls.

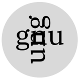
Because some tokens trigger further scanning (e.g. expressions) we also have two dedicated sub tables with scanners: `tokenscanners` and `typescanners` where, when indexed with a token (command) or type you get the appropriate scanner to get a real result. When you look at what is built into ConTEXt you will notice that we often look ahead and then trigger the appropriate scanner. This approach permits to come up with syntaxes that are different than what MetaPost normally does, so for instance brackets and braces can be used to fence parameters and collections, while lists of comma separated numbers can be grabbed that are not part of pairs, triplets, quadruples etc.

19.8 Special helpers

19.8.1 Hashes

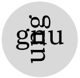
This is typically one of the examples that popped up when Alan Braslau and I were exploring the new possibilities. Due to the way MetaPost implements hashes using Lua might turn out to be more efficient. Here are some examples:

```
\startMPcode
    newhash("foo") ;
    tohash("foo","bar","gnu") ;
    tohash("foo","rab","ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    draw textext(fromhash("foo","bar")) ;
    draw textext(fromhash("foo","rab")) rotated 90 ;
    disposehash("foo") ;
\stopMPcode
```



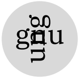
In this example we allocate a hash and afterwards get rid of it. When you don't allocate one it will be automatically allocated. Hashes are persistent, so if you want to be sure you start fresh you'd better create one explicitly. And if you use a large one, you'd better clean up afterwards.

```
\startMPcode
    newhash("foo") ;
    tohash("foo",1,"gnu") ;
    tohash("foo",2,"ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    for i=1 upto 3 :
        if inhash("foo",i) :
            draw textext(fromhash("foo",i))
                rotated ((i-1) * 90) ;
        fi ;
    endfor ;
\stopMPcode
```



Here we check if something is present in a hash. This example also demonstrates that we can use numbers as key. And yes, you can also use boolean keys:

```
\startMPcode
    newhash("foo") ;
    tohash("foo",false,"gnu") ;
    tohash("foo",true,"ung") ;
    fill fullcircle scaled 1cm withcolor "lightgray" ;
    draw textext(fromhash("foo",false)) ;
    draw textext(fromhash("foo",true)) rotated 90 ;
\stopMPcode
```



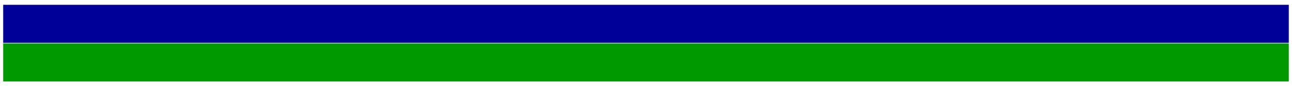
Looking at the implementation of these macros (at the MetaPost end) and functions (at the Lua end) will give you an idea how all these interfaces work together.

19.8.2 Modes

You can query the modes set at the TEX end. You can also check the systemmode.

```
\enablemode[weird]
\startMPcode
    fill fullsquare xyscaled (TextWidth,5mm)
        withcolor if texmode("weird") : "darkblue" else : "darkgreen" fi ;
\stopMPcode
\disablemode[weird]
\startMPcode
    fill fullsquare xyscaled (TextWidth,5mm)
        withcolor if texmode("weird") : "darkblue" else : "darkgreen" fi ;
```

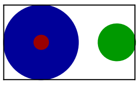
```
\stopMPcode
```



19.8.3 Positions

Keeping track of positions is a core feature and accessible in MetaPosttoo. Here is a somewhat weird example. Positions are always relative to a region, normally the page, but here we provide one via \framed.

```
\framed [region=MyRegion,offset=overlay] \bgroup \hpos {here} \bgroup
    \startMPcode
        fill fullcircle scaled 10mm
            withcolor "darkblue" ;
        draw positionxy("here")
            shifted - positionxy("MyRegion")
            withpen pencircle scaled 2mm
            withcolor "darkred" ;
        draw positionxy("here")
            shifted - positionxy("MyRegion")
            shifted (wdpart positionwhd("MyRegion"),0)
            withpen pencircle scaled 5mm
            withcolor "darkgreen" ;
    \stopMPcode
\egroup \egroup
```



`positionanchor`	string
`positionbox`	path using connector --
`positioncolumn`	numeric
`positioncurve`	path using connector ..
`positiondepth`	numeric
`positionhangafter`	numeric
`positionhangindent`	numeric
`positionheight`	numeric
`positionhsize`	numeric
`positionleftskip`	numeric
`positionllx`	numeric
`positionlly`	numeric
`positionlowerleft`	pair
`positionlowerright`	pair
`positionpage`	numeric
`positionparagraph`	numeric
`positionparindent`	numeric
`positionpath`	path using connector --

`positionpx`	numeric
`positionpxy`	pair
`positionpy`	numeric
`positionregion`	string
`positionrightskip`	numeric
`positionupperleft`	pair
`positionupperright`	pair
`positionurx`	numeric
`positionury`	numeric
`positionwhd`	(wd,ht,dp)
`positionwidth`	numeric

Positioning can be tricky. You really need to make sure that the bounding box of the result is right because when it changes, positions also change you get cyclic runs and quite possible graphics that get larger and larger.

19.8.4 TeX quantities

You can set and get some of TeX's internal quantities:

```
\scratchdimen=100pt \scratchcounter=250 \scratchtoks={okay} \def\Good{good}
\startMPcode
draw textext(getdimen("scratchdimen"))   shifted (0cm,0) withcolor "darkblue"   ;
draw textext(getcount("scratchcounter")) shifted (3cm,0) withcolor "darkred"    ;
draw textext(gettoks ("scratchtoks"))    shifted (6cm,0) withcolor "darkgreen"  ;
draw textext(getmacro("Good"))           shifted (9cm,0) withcolor "darkyellow" ;
\stopMPcode
```

99.62639 250 okay good

Valid getters are `getmacro`, `getdimen`, `getcount` and `gettoks` and their counterparts are `set...` and `setglobal...`. Instead of names you can use numbers for registers, but don't mess up the system ones:

```
\startMPcode
setdimen(2,2*100pt) setcount(2,2*250) settoks(2,"OKAY") setmacro("Good","GOOD")
draw textext(getdimen(2))      shifted (0cm,0) withcolor "darkblue"   ;
draw textext(getcount(2))      shifted (3cm,0) withcolor "darkred"    ;
draw textext(gettoks (2))      shifted (6cm,0) withcolor "darkgreen"  ;
draw textext(getmacro("Good")) shifted (9cm,0) withcolor "darkyellow" ;
\stopMPcode
```

199.2523 500 OKAY GOOD

19.8.5 UTF8

Because we use an utf8 engine we also have MetaPost accepting that encoding. The normal string primitives are unchanged and operate on (ascii) bytes but we have some additional helpers (and more might show up if needed). Here is an example:

```
\startMPcode
string s ; s := "ÀÁÂÃÄÅàáâãäå" ;
draw textext(s)              shifted ( 0cm,0) withcolor "darkyellow" ;
draw textext(utfnum("Â"))    shifted ( 3cm,0) withcolor "darkmagenta" ;
draw textext(utflen(s))      shifted ( 6cm,0) withcolor "darkcyan" ;
draw textext(utfsub(s,3,4)) shifted ( 9cm,0) withcolor "darkblue" ;
draw textext(utfsub(s,6))    shifted (12cm,0) withcolor "darkred" ;
\stopMPcode
```

ÀÁÂÃÄÅàáâãäå 194 12 ÂÃ Åàáâãäå

19.8.6 Checkers

There are a couple of checkers, mostly used in modules. Here's are a few that Alan needs for the node module:

```
\startMPcode
    draw image (
        draw textext(if isarray p[1][2] : "Y__" else : "N__" fi) ;
        draw textext(if isarray p[1]    : "_Y_" else : "_N_" fi) ;
        draw textext(if isarray p       : "__Y" else : "__N" fi) ;
    ) xsized 3cm withcolor "darkred" ;
\stopMPcode
```

YYN

```
\startMPcode
    draw image (
        draw textext(prefix p[1][2]) shifted (10,0) withcolor "darkred" ;
        draw textext(prefix p[1]    ) shifted (20,0) withcolor "darkgreen" ;
        draw textext(prefix p       ) shifted (30,0) withcolor "darkblue" ;
    ) ysized 12mm ;
\stopMPcode
```

p p p

```
\startMPcode
    draw image (
        draw textext(dimension p[1][2]) shifted (10,0) withcolor "darkred" ;
        draw textext(dimension p[1]    ) shifted (20,0) withcolor "darkgreen" ;
        draw textext(dimension p       ) shifted (30,0) withcolor "darkblue" ;
    ) ysized 12mm ;
\stopMPcode
```

2 1 0

```
\startMPcode
    picture p ; p := textext("some text") ;
    path    q ; q := fullcircle scaled 3cm ;
    draw textext(tostring(isobject(p)))                 withcolor "darkgreen" ;
    draw textext(tostring(isobject(q))) shifted (50,0) withcolor "darkblue" ;
\stopMPcode
```

```
true    false
```

19.8.7 Key-value interfaces

There are plenty of examples in the `mp-lmtx.mpxl` *file and more will be added. Just make sure you create your own unique namespace and don't use the ones that ConTEXt uses (like* `lmt_`*).*

20 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
```