Drawing graphs with MetaPost

Alan Braslau

The graphical representation of data is discussed.

1. Preamble

WARNING: The following is both *fact* and *fiction*. Fact, in that all of the graphics presented here are produced using MetaPost. Fiction, in that it describes the use of MetaPost version ≥ 1.760, available via `svn` but not yet included in the ConTEXt standalone distribution. Furthermore, the `graph` package described here will be re-written, simplified, and extended, taking advantage of this new version of MetaPost. This has yet to be done.

It may appear that we are going about this somewhat backwards, as the present article is, in fact, an attempt to document the new `graph` package. It is indeed strange to begin by writing the manual for a (partially) fictitious package *before* the macros have been developed. But not really, for this should serve as a sort of specification, following which the programming should be easy. As a bonus, the documentation will thus already exist, rather than having to take on the task of its creation!

Finally, this article is intended to provoke comment, criticisms, and suggestions, for it is a work in progress (and far from complete). As the package is rewritten, things will change. The reader is thus warned.

2. Introduction

This document describes a graph-drawing package that has been implemented as an extension to the MetaPost graphics language. It is based upon an original package written by John D. Hobby, also the creator of MetaPost. With the evolution to MetaPost v2, calculations may now be performed in double precision floating-point arithmetic at no performance cost due to progress in modern processors. This greatly simplifies the manipulation of data and the implementation of the graph-drawing package.

The aim of the package is the drawing of clean, graphical representation of data. This is made possible though its integration into a high-quality publishing system, leading to a coherent use of style, typeface, and scale. The TEX text-processing system created by Donald Knuth provides such a structure, and the MetaPost graphics language allows a mathematically precise drawing of figures. The world of publishing has evolved from the use of the PostScript printing language to the widespread use of the `pdf` Portable Document Format as well as standardized markup languages such as `xml`. TEX has likewise evolved to the direct production of `pdf` documents; its current 'engine' under active development is `luatex` that includes extensions integrating the Lua scripting language. The MetaPost graphics engine (`mpost`), a stand-alone system producing drawings encoded in the PostScript language or, alternately, Scalable Vector Graphics (SVG) has been integrated into `luatex` through the `mplib` library. The one document producing system integrating TEX layout and MetaPost graphics is ConTEXt, a coherent and complete TEX-based package.

The present document describes the use of a new `mpgraph2` macro package as integrated in ConTEXt through `luatex`. As such, it will eventually become part of the

ConTEXt suite. The macros may also be used in standalone MetaPost producing figures in encapsulated PostScript format that can then be integrated as external graphics in other documents produced with LaTEX, for example.[1]
An important starting point is an introduction to the MetaPost graphics language. This will not be covered here. A good introduction is the MetaPost manual (`mp-man.pdf`) that is generally distributed with MetaPost; it is also available on CTAN. This document should be considered 'required reading' before attempting to do anything using MetaPost. Another good introduction is the MetaFun manual (`metafun-s.pdf` and `metafun-p.pdf` for the screen and printed versions, respectively) which is also available as a printed and bound volume. MetaFun integrates MetaPost into the core of ConTEXt with specific extensions. Other good introductions and tutorials have been written and many links can be found on `http://www.tug.org/metapost.html`.
The present documentation attempts to integrate notions and ideas (and even some prose) as presented in the documentation to the original MetaPost graphics package (`mpgraph.pdf`).

3. Choice of tools

The ConTEXt typesetting system uses `luatex` as a processing back-end. Graphical representation of data can be implemented through ConTEXt macros, as `lua` functions, as a MetaPost package, or through a combination of these. From a user's point of view (as opposed to a programmer's point of view), it is preferable to 'see' only one language, thus either TEX (ConTEXt) or MetaPost (for graphics).[2] The choice of a ConTEXt package makes sense for standardized representations of data; simple pie charts or histograms might be good examples[3] and several 'modules' in this direction have been written. However, few graphics can indeed be standardized, and the graphical user would want to retain more control through the use of the entire MetaPost language. This is the choice of the present implementation. Here, graphics are created in pure MetaPost (of course, text strings *can* be formatted in TEX through the use of the `textext()` function or the `btex etex` construction).

4. Fundamental notions

Graphics are drawn in MetaPost in a 2D Cartesian coordinate system describing a region of the printed page or the display screen. Its basic unit is the PostScript point (`bp`: 1/72 of an inch) and may be scaled to centimeters (`cm`), millimeters (`mm`), inches (`in`), points (`pt`: 1/72.27 of an inch), picas (`pc`), ciceros (`cc`), Didot points (`dd`), or your own scale (by convention, `u`), as needed. Data to be represented by a graph may be in

[1] Another good graphics package for TEX is pgf/TikZ and the associated pgfplots package. It is distributed with an excellent documentation and tutorial. ConTEXt has always been well supported, however the syntax and philosophy of the package is best-tuned for LaTEX users.

[2] Lua is a clean, interpreted programming language. Unfortunately, a non-programmer can be left completely dismayed when faced with even a simple Lua routine.

[3] A ConTEXt macro package for some standardized representations of data would almost certainly use MetaPost as a graphical engine, but this would be hidden from the user.

any other space or coordinate system, to be then transformed to the display space of the MetaPost graphics.

Example 1.

We begin by reserving a canvas where the data is to be displayed. This is a rectangular frame having two dimensions: a width and a height, and whose lower left-hand corner is located at the origin of the MetaPost drawing space.
In ConTEXt, one might write:

```
\usemodule [graph]
\starttext
\startMPcode
draw
  begingraph(5cm,5cm) ;
  gdraw (0,0)--(1,1) ;
  endgraph ;
\stopMPcode
\stoptext
```

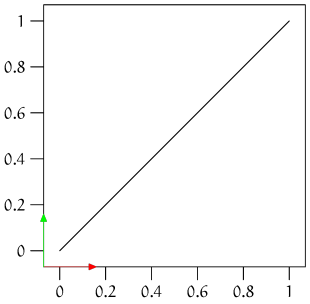

Figure 1: A first graph

Equivalently, one could write:

```
input graph ;
beginfig(1) ;
draw
  begingraph(5cm,5cm) ;
  gdraw (0,0)--(1,1) ;
  endgraph ;
endfig ;
end
```

The use of standalone `mpost` produces an encapsulated PostScript file (`eps`). (In the following, we will concentrate solely on MetaPost as integrated in the core of ConTEXt through `mplib`, forgetting about standalone `mpost`.)
The result is shown Figure 1. Also drawn are the vectors `(1cm,0)` in red and `(0,1cm)` in green (the corresponding MetaPost code is not shown here). Note that their origin is at the corner of the canvas, not the origin of the graph data. The object drawn by the `begingraph()` statement is collected until 'shipped-out' following the `endgraph` closing statement; it is a picture object that may be operated on, for example `shifted`.

The scale transforming from data space to the drawing canvas is automatically determined by the extrema of the data — lower left and upper right corners. The drawing of several data sets (through several `gdraw` statements) would all contribute to the automatic determination of the XY-scale. Conversely, the scale may be

`draw begingraph(`*w*`,`*h*`)`	Begin a new graph with the frame width and height given by numeric parameters *w* and *h*.
…	
`endgraph`	End a graph and return the resulting picture.
`setcoords(`*x*`,`*y*`)`	Set up a new coordinate system as specified by the labels *x* and *y*. Label values are `±linear`, `±log`, `±ln` and `±sqrt`.
`setrange((`*x1*`,`*y1*`), (`*x2*`,`*y2*`))`	Set the lower and upper limits for the current coordinate system. Each (*x*,*y*) can be a single pair expression or two numeric or string expressions.
`gcoord(`*x*`,`*y*`)`	Takes a pair of graph coordinates and returns a pair of MetaPost coordinates.

Table 1: General graph command summary

explicitly set, through the use of the `setrange()` statement taking as parameters two pairs of lower-left and upper-right coordinates.[4] Coordinates set explicitly through `setrange()` are taken to be exact. The use of the anonymous variable `whatever` for any value rounds the range up or down according to the extrema of the plotted data.[5] Similarly, the transformation from data space to drawing space may be according to some (non-linear) function. These may be set using the `setcoords()` statement, taking two labels, one for the abscissa and a second for the ordinate. These labels may be the keywords '`linear`', '`log`','`ln`' or '`sqrt`';[6] a negative coordinate type (i.e. `-linear`, `-log`) makes the axis run backwards, that is right to left or top to bottom.[7] The function `gcoord()` returns a value in MetaPost drawing coordinates given a value in graph-space coordinates.
A 3D representation of data will be discussed later. In this situation, the `setrange()`, `setcoords()` and `gcoord()` commands would all require three parameters, of course.

5. Drawing axes

By default, a frame around the canvas is drawn, and the abscissa and ordinate are marked and numbered on the bottom and on the left. This may be further controlled

[4] Given the flexibility of the MetaPost language, the parameters may also be written as a series of four numbers, dropping the pair parenthesis, as in `setrange(0,0,whatever,whatever)`, here fixing the data origin to coincide with the drawing origin. However, the form maintaining the parenthesis may be clearer, avoiding confusion as to the order of the parameters: `setrange((0,0),(whatever,whatever))`.

[5] A trick that can be used to set a range that will then be rounded is to 'plot' a path of extrema with a `nullpen`, as in: `setrange ((whatever,whatever), (whatever,whatever)) ; gdraw (0,0)--(pi,pi) withpen nullpen ;` Of course, subsequent data to be plotted must be contained within the set extrema; otherwise, they will eventually set new extrema.

[6] '`linear`' and '`log`' are currently the only possible choices for coordinate systems.

[7] A further type might be the single label '`polar`', interpreting the data as pairs of (radius,angle).

using the gaxis() command.[8] It takes a mandatory suffix (lft, rt, top, bot, x, y) and a list of parameters. The default settings are given Table 2.

```
setcoords(linear,linear) ;
setrange((whatever,whatever),(whatever,whatever)) ;
gaxis.bot(line,tick.bot,numbers) ; gaxis.top(line) ;
gaxis.lft(line,tick.lft,numbers) ; gaxis.rt(line) ;
```

Table 2: Default settings

Example 2.

Drawing any one axis suppresses the automatic drawing of the other three axes, which must then be explicitly drawn, if needed. Multiple calls may be made, changing pens, colors, etc.

```
\startMPcode
numeric w ;
w := \the\marginwidth ;
draw begingraph(w,w) ;
  gdraw (0,0)--(1,1) ;
  gaxis.bot(grid) dashed evenly withcolor
                                    .85white;
  gaxis.bot(line,numbers) ;
  gaxis.top(line) ;
  gaxis.lft(grid) dashed evenly withcolor
                                    .85white;
  gaxis.lft(line,numbers) ;
  gaxis.rt(line) ;
  endgraph ;
\stopMPcode
```

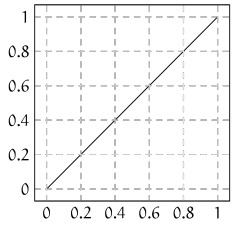

Figure 2: Controlling axes

Here, the dimensions of the graph are dependent on the layout. Because \textwidth is stored in TEX dimension registers, it must be prefixed with \the expanding to a dimension (in pt).

Example 3.

The graph axes can take alternate forms. Here, we draw just the bottom and left axis along with a grid scale This example further demonstrates the drawing of a functional form, here a simple sinusoid, as will be further explained later.

[8] This is a new feature, replacing the autogrid(), grid() and frame commands.

```
\startMPcode
vardef FUNC(expr x) = sin(x*pi) enddef;
draw begingraph(w,w) ;
  setrange((0,whatever),(2,whatever)) ;
  gdraw FUNC withcolor red ;
  gaxis.x(arrow,numbers, major4,minor2,
          skipfirst);
  glabel.top(btex x etex, (2,0)) ;
  gaxis.y(arrow,numbers, major4,minor2,
          skipfirst, skiplast) ;
  glabel.rt(btex y etex, (0,1)) ;
  endgraph ;
\stopMPcode
```

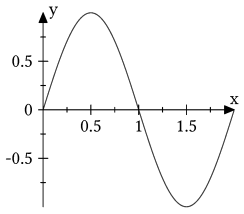

Figure 3: XY axes

Note that it is good practice in the graphical representation of functions to draw the axes with arrows indicating the positive direction; Both axes should be labeled, as should be the origin.

The labeling commands are closely related to a set of similarly named commands in plain MetaPost; They can be followed by an *option list* of usual MetaPost drawing options.

`glabel.`*suffix*`(`*p*`,` *location*`)`	If *p* is not a picture, it should be a string. Typeset it using `defaultfont`, then place it near the given location and an optional offset as specified by the label *suffix*. The *location* can be x and y coordinates, a pair giving x and y, a numeric value giving a time on the last path drawn, or `OUT` to label the outside of the graph frame.
`gdotlabel.`*suffix*`(`*p*`,` *location*`)`	This is like `glabel()` except it also puts a dot at the location being labeled.
`gaxis.`*suffix*`(`*list*`)`	This draws a graph axis, where the mandatory *suffix* is one of `lft`, `rt`, `top`, `bot`, `x`, `y` and determines which axis is to be drawn (`x` and `y` draw an axis passing through the data origin). *list* determines options and may contain: `line`, `arrow`, `numbers`, `skipfirst`, `skiplast`, `skipzero`, `grid`, `tick.`*suffix*, `major`*M*, `minor`*N* (setting the number *M* or *N* major or minor tick intervals, otherwise automatically determined. Major tick intervals are labeled, minor tick intervals are hashed with a shorter line.)

Table 3: Axis and labeling command summary

6. Drawing data

But we have yet to draw any real data! Before looking into importing data, we will first cheat and simulate data through calculation.

Example 4.

We define a path p through a Gaussian function and a modified path q with added noise, simulating data.

```
\startMPcode
path p ; p := for i=1 upto 49:
    if i>1 : -- fi
    hide(x := i/25-1 ;) (x,1000*exp(-4x*x*log(2)))
  endfor ;
path q ; q := for i=0 upto length p:
    if i>0 : -- fi
    (point i of p
     shifted (0,normaldeviate*sqrt(ypart (point i of p))))
  endfor ;
\stopMPcode
```

The Gaussian path is drawn in figure 4 as a continuous line and the 'data' are plotted using a pre-defined symbol.

```
\startMPcode
draw begingraph(w,4w/3) ;
  setcoords(linear,log) ;
  setrange((-2,.01), (+2,2000)) ;
  gaxis.x(arrow,numbers,skiplast) ;
  glabel.bot(btex x etex,(xmax,ymin)) ;
  gaxis.y(arrow,numbers,skipfirst) ;
  glabel.rt(btex y etex, (0,ymax)) ;
  gdraw p withcolor red ;
  gdraw q plot plotsymbol(1,blue,.4) ;
  endgraph ;
\stopMPcode
```

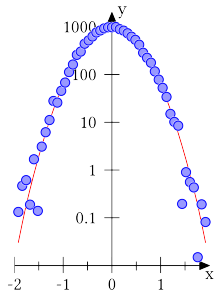

Figure 4: A Gaussian distribution

6.1 Plotting symbols

The function `plotsymbol()` takes three parameters: a number specifying the shape, a color, and a fill color (or number, interpreted as a shade of the drawing color). The number specifying the shape is 0 for a dot, 1 for a circle, 2 for a vertical bar, 3 for a triangle, 4 for a square, 5 for a pentagon, … 9 for a nonagon. Variants are also defined: adding 10, 20, or 30 to the index give rotated or modified symbols. Not all possibilities

are defined—-only those that make sense. The symbols are illustrated in figure 5. The size of the symbol is given by the numerical parameter `sym_size`; By default, it is defined as half of the font size of the `defaultfont`, but may be freely redefined.
The symbol shapes are stored in an array of closed paths: `sym_`.
When plotting data sets, it is useful to choose colors from a palette of easily identified standards. We thus define an array of colors co and its corresponding array of string names cn with indices running from 0 to 9, according to the rainbow of color codes (used to identify electrical resistances):
0:black 1:brown 2:red 3:orange 4:yellow 5:green 6:blue 7:violet 8:gray 9:white

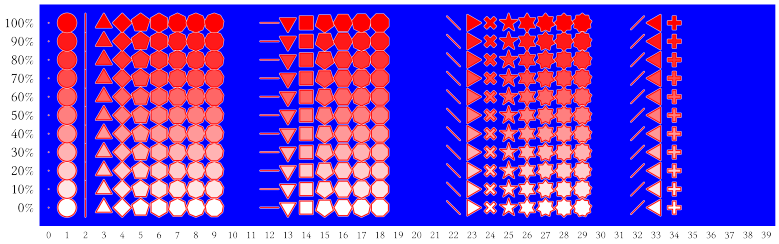

Figure 5: Plot symbols — `plotsymbol(`*index,red,shade*`)`.
The abscissa is the symbol index and the ordinate is the fill shading.

6.2 Plotting mathematical functions

In the previous example, a path of points following a Gaussian function was calculated and manipulated. Earlier, a sinusoidal function was drawn. The command `gdraw` can draw a path, data taken from a file (see below), or a functional form returning an ordinate given an abscissa.[9] In this case, the range of the plot must be determined, either through the previous drawing of a path of data or else through an explicit use of `setrange()`. The density of points is determined by the parameter `ncalc`, spaced evenly in drawing space, thus not necessarily linearly in data space.
As this feature has not yet been programmed, the implementation may well be different. For example, it may prove advantageous to provide a command `calculate_path()` returning a path to be drawn given a function `FUNC()`, a more sophisticated implementation of the basic algorithm:

```
for i=1 upto ncalc :
       if i>1 : -- fi
       hide (x := i/(ncalc-1)*[xmin,xmax] ;) (x,FUNC(x))
endfor
```

Plotting functions is an important feature of a graph package. Furthermore, calculating functions from a set of data needs to be implemented. Examples that come to mind that can be exactly calculated are minimum, maximum, mean, variance, polynomial, etc.. More sophisticated would be non-linear least-squares optimization or maximum entropy determination. Other standard statistical treatments could also be envisioned.

[9] Alternately, the drawing of a multi-valued function may be better handled by a function returning an abscissa given an ordinate. The author accepts that this point needs to be further developed.

6.3 Reading data files

Real data can be collected from a variety of sources and will be stored in a file or a collection of files in *some* format. The simplest such format is, commonly, columns of data, typically numbers represented as character digits or even strings. Columns may be delimited by 'white space' (a series of spaces and tabulations), by a single tabulation character, or by a separation character such as a comma or a semicolon (this is often called CSV). Alternately, data may be stored in some 'binary' format or in a more complicated hierarchical format (one example is 'HDF5').
Data stored in columns can be easily read line by line (a 'record') and parsed into columns ('fields'). The function `gdata(filename,variable,commands)` opens a data file '*filename*', processing each record, setting the line number to the variable *i*, assigning fields as strings to *variable*1, *variable*2, … up to the number of fields per line (assigned to the variable *j*), terminating the 'array' with the null string; for each record the *commands* are executed, allowing one to process the data as desired.[10]

Example 5.

Consider the following data:

```
month  2002  2003  2004  2005  2006  2007  2008  2009  2010  2011
1      0     462   285   370   877   667   523   554   436   845
2      0     476   177   523   446   538   425   712   747   760
3      0     579   120   935   509   455   421   1106  1052  1079
4      2     355   113   649   885   450   1076  1024  846   558
5      0     293   216   442   767   336   456   961   746   736
6      0     187   349   499   719   356   723   623   729   835
7      0     180   242   635   774   431   880   518   628   700
8      0     293   437   516   762   745   683   575   846   629
9      0     626   262   543   539   461   890   562   790   489
10     171   328   299   406   506   332   746   715   945   486
11     277   142   459   771   347   496   805   999   940   0
12     276   252   365   905   403   766   576   525   782   0
total  726   4173  3324  7194  7534  6033  8204  8874  9487  7117
```

This data was taken from the ConTEXt mailing list, showing the number of messages exchanged each month. The data file '`context.dat`' is read and the data placed in the paths `p0`—`p10`.

```
\startMPcode
path p[] ;
numeric year[] ;
gdata("context.dat",s,
```

[10] This approach is similar to that implemented in the interpreted program awk. Indeed, it could interesting, following awk, to provide some pattern-matching mechanism.

```
  for j=0 upto 9 :
    if s1="month" :
      year[j] := scantokens(s[j+2]) ;
    elseif s1="total" :
      augment.p10(year[j], scantokens(s[j+2])) ;
    else :
      augment.p[j](scantokens(s1), scantokens(s[j+2])) ;
    fi
  endfor
) ;
\stopMPcode
```

The standard MetaPost function `scantokens()` converts a string into its numerical value. The function `augment.variable(coordinates)` adds to the path *variable* the pair given by *coordinates*. The paths may then be plotted.
First, we illustrate the growth in postings, year by year.

```
\startMPcode
draw begingraph(w,w) ;
  glabel.bot(btex year etex, OUT) ;
  glabel.lft(btex postings per year etex
             rotated 90, OUT) ;
  gdraw p10 ;
endgraph ;
\stopMPcode
```

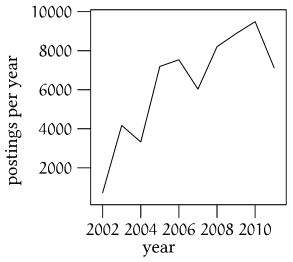

Figure 6: ConTEXt mailing list postings by year

We observe that the number of postings has grown by about a factor of 5, approaching around 10 000 postings per year (at the present date, the year 2011 is only 10/12 complete). Note that this level corresponds to about 26 postings per day, on average.

```
\startMPcode
draw begingraph(w,w) ;
  glabel.bot(btex month etex, OUT) ;
  glabel.lft(btex postings per month etex
             rotated 90, OUT) ;
  for j=1 upto 9 :
    gdraw p[j]
    withcolor co[j-1] ;
  endfor
endgraph ;
\stopMPcode
```

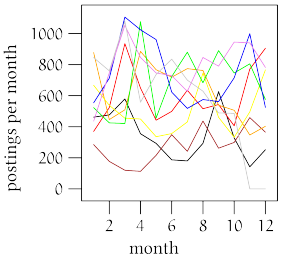

Figure 7: ConTEXt mailing list postings by month, 2003—2011 (black—gray)

Notice the use of the color array co as defined in the previous section. No clear pattern can be distinguished in the postings by month.
The postings were further analyzed, producing the following cumulative data:

```
Sun     6432
Mon     9811
Tue     10496
Wed     10724
Thu     10377
Fri     9291
Sat     5535
```

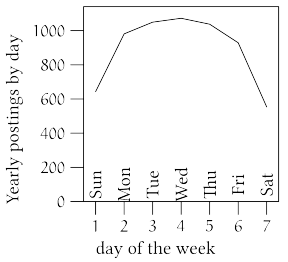

Figure 8: ConTEXt mailing list average yearly postings by day of the week

showing that the midweek traffic is about double that on weekends.

```
\startMPcode
draw
  begingraph(w,w) ;
  setrange ((whatever,0),(whatever,whatever)) ;
  path p ;
  gdata("context2.dat",s, augment.p(i,scantokens(s2)/10) ;
  glabel(textext(s1) rotated 90, (i,100)) ; ) ;
  glabel.bot(btex day of the week etex, OUT) ;
  glabel.lft(btex Yearly postings by day etex rotated 90, OUT) ;
  gdraw p ;
endgraph ;
\stopMPcode
```

Example 6.

Data showing the daily electricity consumption in France over the last sixteen years is publicly available. The values, in MWh, are too large to be represented using standard MetaPost scaled-point arithmetic; We plot this data here rescaled in GWh, somewhat easier to comprehend as well (knowing that the total theoretical electrical power production capacity is about 105 GW — nuclear, hydroelectric, oil, coal, natural gas, solar, wind, tidal, … — or about 2500 GWh per day).

```
\startMPcode
draw
  begingraph(8cm,6cm) ;
  setcoords(linear,linear) ;
  setrange(1995.5,0,2012,2500) ;
```

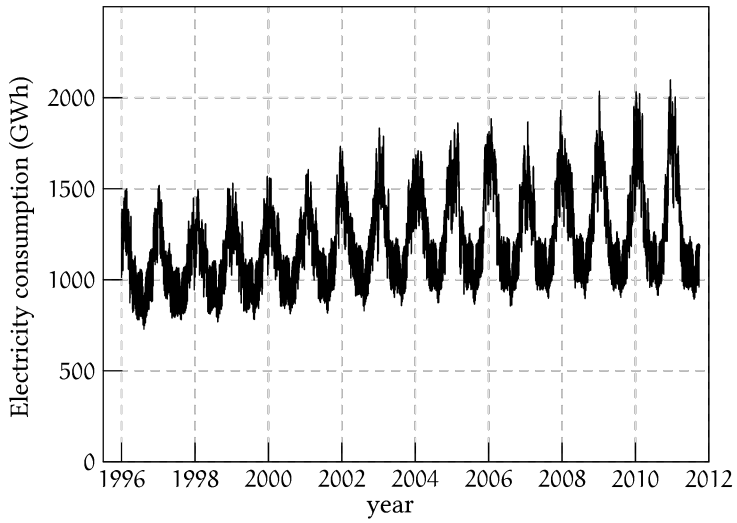

Figure 9: Electricity consumption in France (data RTE)

```
  glabel.bot(btex year etex, OUT) ;
  glabel.lft(btex Electricity consumption (GWh) etex rotated 90,
             OUT);
  gaxis.bot(grid) dashed evenly withcolor .85white ;
  gaxis.rt (grid) dashed evenly withcolor .85white ;
  gaxis.bot(line,tick.top,numbers) ;
  gaxis.top(line) ;
  gaxis.lft(line,tick.rt, numbers) ;
  gaxis.rt(line) ;
  path p ;
  for year = 1996 upto 2011 :
    days := if ((year mod 4) = 0) :
              366
            else :
              365
            fi ;
    gdata("Historique_consommation_JOUR_" & decimal year & ".csv",
          s, augment.p(year+(i-1)/days, scantokens(s2)/1000) ; ) ;
  endfor
  gdraw p ;
endgraph ;
\stopMPcode
```

This shows a steady increase (about 30%) in the electricity consumption over the past 16 years.
The annual variation can be compared year by year. As there is also a strong weekly variation, the data needs to be offset by the starting day of the week (1st of January):

```
\startMPcode
draw
  begingraph(8cm,6cm) ;
  setcoords(linear,linear) ;
  setrange(9,0,372,2500) ;
  glabel.bot(btex day etex, OUT) ;
  glabel.lft(btex Electricity consumption (GWh) etex rotated 90,
             OUT);
  gaxis.bot(grid) dashed evenly withcolor .85white ;
  gaxis.rt (grid) dashed evenly withcolor .85white ;
  gaxis.bot(line,tick.top,numbers) ;
  gaxis.top(line) ;
  gaxis.lft(line,tick.rt, numbers) ;
  gaxis.rt(line) ;
  path p[] ;
  for year = 1996 upto 2011 :
    day := year mod 7 ;
    if ((year mod 4) = 1) : day := day + 1 ; fi
    gdata("Historique_consommation_JOUR_"& decimal year & ".csv",s,
          augment.p[year-1996](day+i, scantokens(s2)/1000) ; ) ;
    gdraw p[year-1996] withcolor co[(year mod 8)+1] ;
  endfor
endgraph ;
\stopMPcode
```

See figure 10 on the next page.
This programming trick could be simplified through the use of proper date parsing functions that need to be introduced to the graphing package, as time is an important variable in much data.
The preceding data might be better viewed in 3D. Figure 11 (next page) was indeed drawn using MetaPost, but not using `graph` macros, nor using one of several 3D macro packages that were discussed at the ConTEXt user's meeting. Rather, here I cheated and used my old-time favorite graphics program called `cplot` (written by Gerry Swislow).
The 3D representation of data in MetaPost can be handled using different approaches: one is the use of arrays of 'colors' — triplets; a second is the use of three coupled arrays of numbers (using the suffix mechanism). As MetaPost has no true array mechanism (array elements are simply allocated variables sharing a common name and an appropriate syntax to handle numerical labels), these two approaches might prove to be inefficient for a large set of data. A third approach, the possibility to define paths of triplets, would require MetaPost development. Although this would be conceptually the most satisfying, in practice it may or may not be possible. The handling of 3D data (and drawing 3D objects, in general) remains an open question.[11]

[11] For example, should formatted text be projected, as is shown figure 11?

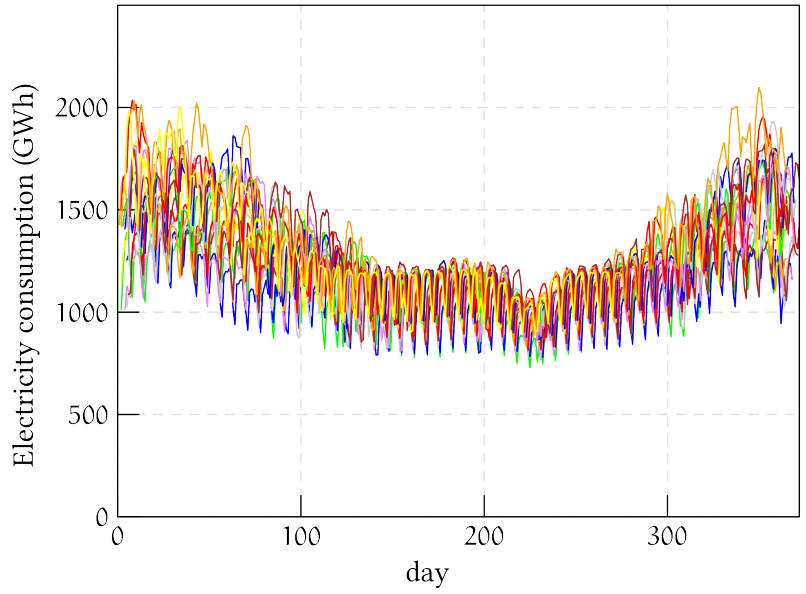

Figure 10: Yearly variation in the electricity consumption

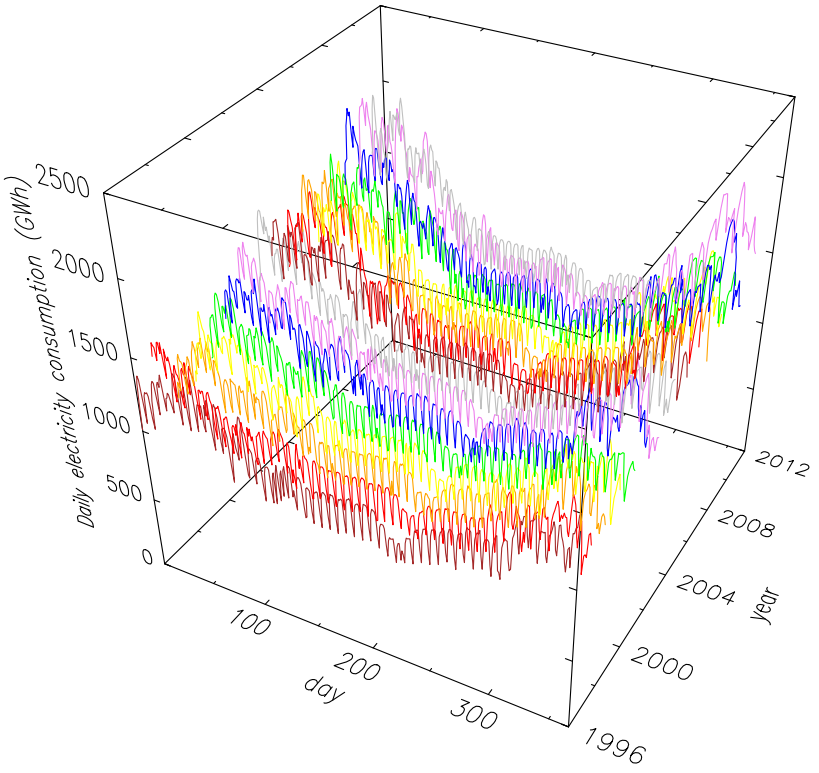

Figure 11: Yearly variation in the electricity consumption (3D)

All of the drawing commands can be followed by an option list. In addition to the usual MetaPost drawing options, the list can contain a `plot picture` clause to plot a specified picture at each data point. The drawing commands are closely related to a set of similarly named commands in plain MetaPost.

`gdraw` *p*	Draw path *p*, or if *p* is a string, read coordinate pairs from file *p* and draw a polygonal line through them.
`gfill` *p*	Fill cyclic path *p* or read coordinates from the file named by string *p* and fill the resulting polygonal outline.
`plotsymbol(`*shape*`,` *draw*`,` *fill*`)`	Returns a plot symbol as a picture to be drawn. Takes a *shape* given as a number related to a polygonal dimension (shown figure 5) or as an arbitrary closed path, that is drawn using the color *draw* and filled using the color *fill*. If the fill color is a number, it is taken as a shade between the background and the *draw* color: `fill[background,draw]`.
`gdata(`*f*`,` *variable*`,` *commands*`)`	Read the file named by string *f* and execute *commands* for each input line using the *variable* as an array to store data fields.
`augment.`*variable*`(`*coordinates*`)`	Append *coordinates* to the path stored in *variable*.
`scantokens` *string*	A MetaPost primitive. Converts a string to a token or token sequence. Provides string to numeric conversion, etc.

Table 4: Data handling command summary

6.4 Error bars

It is common to indicate the uncertainty in recorded data through the drawing of error bars for the abscissa, the ordinate, or both; Sometimes, an error 'ellipse' is drawn. One can easily draw error bars using MetaPost commands within the `gdata()` function. However, a standard mechanism needs to be developed, for example, given a `path` of data and an associated 'path' of error values (either pairs of *x, y* errors, *y-, y+* errors, etc.). This need will also be addressed.

7. Conclusions

The MetaPost `graph` package is to be re-written taking advantage of the floating-point arithmetic implemented in MetaPost v2. This will allow it to be simplified and expanded. The present article is the beginning of a reflection on a proposal for such a task. It is currently necessarily incomplete. To that end, comments and suggestions are most welcome.