Simple drawings with METAFONT

Zdeněk Wagner

June 6, 1995

Introduction

This contribution explains how to use METAFONT for simple drawings. To make the text shorter, some commands are not described. The reader is kindly asked to look into this source to see how it was done.

Nobody can guarantee that METAFONT is exactly what you need. the simple rules say:

- Use TEX where TEX is good.
- Use METAFONT where METAFONT is good.
- Use other tools where other tools are good.

Here you can see examples where METAFONT is good.

Why not other packages?

Though it might look strange I prefer explaining it here. The selection of tools is more (or less) a matter of personal taste. But there should be some reasoning behind it. If you want to make simple drawings, you may choose either MFpic or a similar package or directly METAFONT. In either case you must learn some new commands. However, MFpic supports only a subset of METAFONT. Later, if you need more complex pictures, you have to learn a new tool. With METAFONT it's a bit easier. You just learn some more commands.

It's not a good practice to reject everything what has been done. you can find files with METAFONT macros which can be used in a similar way as LATEX styles. This can make life much easier.

Principle of METAFONT pictures

The principle is to make a new font where a picture is some "character". When we want to place the picture into the document, we change the font and type appropriate character. If the picture is too large or too complex, it is better to divide it into several characters and overlay them with `\llap` or `\rlap` commands or simply place the characters in the correct order (we will see it later).

Initial commands

It is clever not to use absolute dimensions in the drawings. If we measure everything as a multiple of a unit length, we can easily scale the whole font. The unit length should be specified in sharp units (designated with #) which are device independent. We must then convert it into device dependent number of pixels by calling `define_pixels`. To do this, METAFONT must know the properties of the output device. To set everything up, you should call `mode_setup` at the very beginning and supply the correct mode when you call METAFONT.

It may be interesting to see on screen how METAFONT is drawing the picture. It is accomplished by `screenstrokes`. The beginning of METAFONT source file may therefore look as:

```
% This is drawing.mf, an example file
% (C) Z. Wagner, 23 Jan 1993
% This file must not be distributed separately. It is an integral
% part of drawing.tex. It may be placed at any computer in case
% drawing.tex is available at an appropriate directory.
mode_setup;
u# := 1.0mm#;
define_pixels(u);
screenstrokes;
```

Assignments and equations

As the title says, METAFONT can solve equations. Thus you can write ($3a$ is a shorthand for $3 * a$)

```
3a + b = 5;
2a - 3b = 7;
```

After reading these equations the values of a and b are fully defined.

In the previous section we used `:=` which denotes assignment. If you now say

```
a := 13;
```

it will instruct METAFONT to forget whatever value a might have had and assign 13 ti it. In the previous case, when reading the equation $2a - 3b = 7$, METAFONT already knows that $a = (-b + 5)/3$ and these together enable the evaluation of a and b. This is the difference between assignments and equations.

Points, coordinates and simple curves

The position of points are specified using Cartesian coordinates. since METAFONT works inside a plane, we need a pair of numbers, namely the x and y coordinates.

The METAFONT character is usually defined by many points. It is therefore comfortable to index them. METAFONT uses convention known from programming languages, i.e. the index is placed in square bracket as in z[7k-6]. It would be very tedious to write z[3]. Therefore METAFONT offers a shorthand: one simply types z3.

When defining the position of any point, you can either use the pair variable in the equation or you can access the x and y coordinates directly. Thus

```
z3 = (7.3u,-13.4u);
```

is equivalent to

```
x3 = 7.3u;  y3 = -13.4u;
```

whereas u was defined above.

Later in the METAFONT definitions we will use commands as $z3 = t[z1, z2]$. This means that z3 lies on the straight line defined by z1 and z2. If $t = 0$, z3 is equivalent to z1. Equivalence between z2 and z3 holds if $t = 1$. In case $t = \frac{1}{2}$, z3 lies in the middle between z2 and z3.

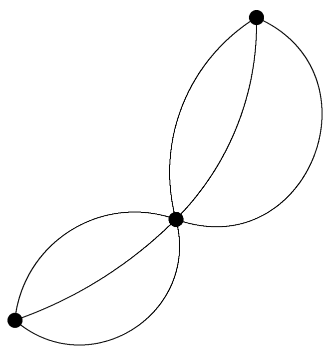

Figure 1: Simple curves

The drawings are composed of Bezier curves. To draw a Bezier curve through z1, z2, z3, we simply write:

```
draw z1..z2..z3;
```

Sometimes we need better control. It may be achieved by specifying a direction at one or more points. This was illustrated in fig.Before we write the METAFONT program for the curves, we must say something about other commands.

The `beginchar` commands starts the definition of a character. The first parameter says what character it is to be assigned to. The next parameters specify the width, height, and depth (how far below the baseline it should extend), respectively. The dimensions must be given in sharp units unless they are zero. To draw something we must first pickup a pen of appropriate thickness. To make a dot wit a pen, we use `drawdot`. The program for the character ends with `endchar`. Now we can look at it:

```
beginchar("A",50u#,50u#,0);
z1 = (5u,3u);  z2 = (37u,43u);
```

```
x3 = 1/3[x2,x1];  y3 = 1/3[y1,y2];
pickup pencircle scaled .4pt;
draw z1..z3{dir 45}..z2;
draw z1..z3{dir 105}..z2;
draw z1..z3{dir -20}..z2;
pickup pencircle scaled 2u;
drawdot z1;  drawdot z2; drawdot z3;
endchar;
```

The command `flex(z1,z2,z3)` draws a Bezier curves through z1, z2, z3, where the direction at z2 is equal to the slope of the straight line from z1 to z3. Make it as your own exercise.

Changing curvature with tension

Bezier spline is a cubic curve. Therefore, you need four points for full specification. However, every Bezier segment in previous examples was defined only by two endpoints. It means that METAFONT has its own algorithm for finding the remaining two points. If you want to control the curvature, you must have the possibility to influence this algorithm. One way is to use `tension`.

You can define tension at any point of any segment. The general syntax is

```
z1..tension a and b..z2
```

If $a = b$, we can simplify this to

```
z1..tension a..z2
```

The simple case `z1..z2` is a shorthand for[1]

```
z1..tension 1..z2
```

In the next example, we will need some more definitions. You should already understand the first two lines. The next line defines some parameters which will be used later. Then we declare the array of paths to be drawn, the array of cnt and pen widths which are both numeric variables. Then we assign some values to them.

```
height#=50u#;  width#=50u#;
define_pixels(height,width);

relsh:=.005;  tens:=3;

path p[];
numeric cnt[], penw[];
penw0=3pt;  penw1=1pt;  penw2=.3pt;
cnt1=.95;   cnt2=.92;
```

[1]This is not the whole truth but you can live with this small lie. If you wish to know more details, you have to study METAFONTbook.

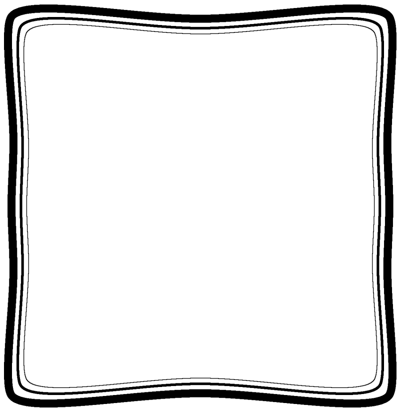

Figure 2: Frame

Afterwards we have to specify a number of points and the three curves. We will use a predefined constant *origin* which means (0,0). To simplify the task we also use loops. The loop starts with the `for` token and carries out the commands up to `endfor` for the specified values of the loop control variable. In the firs loop, when $k = 1$, the inner loop defines points $z20, z21, \ldots, z32$. The token `shifted` means that the point is shifted by the specified vector. The vectors `right`, `left`, `up`, `down`, are the unit vectors in the named direction. The token `scaled` means scaling (multiplication) of the vector. In the equations below it defines the length of the vector.

The definition of paths (curves) end with `cycle`. This means that the curve is cyclical. If you say

```
draw z1..z2..z3..z1;
```

the curve will most probably have a sharp edge at z1. You must say

```
draw z1..z2..z3..cycle;
```

in order to make the curve smooth.

```
z0=origin;  z1=(0,height);  z2=(width,height);  z3=(width,0);
z5=.5[z0,z1];  z6=.5[z3,z2];
z7=.64[z0,z5] shifted (right scaled (relsh*width));
z8=.64[z1,z5] shifted (right scaled (relsh*width));
z9=.64[z2,z6] shifted (left scaled (relsh*width));
z10=.64[z3,z6] shifted (left scaled (relsh*width));
z11=.5[z1,z2] shifted (down scaled (relsh*height));
z12=.5[z0,z3] shifted (up scaled (relsh*height));
z13=(.5width,.5height);
for k:=1 upto 2:
  m:=20k;
  for j:=0 upto 12 :
```

```
    z[j+m]=cnt[k][z13,z[j]];
  endfor;
endfor;
for k:=0 upto 2:
m:=20k;
p[k]=z[m]..tension tens and 1..z[m+7]..z[m+5]..z[m+8]..tension 1 and tens..
  z[m+1]..tension tens and 1..z[m+11]..tension 1 and tens..z[m+2]..
  tension tens and 1..z[m+9]..z[m+6]..z[m+10]..tension 1 and tens..z[m+3]..
  tension tens and 1..z[m+12]..tension 1 and tens..cycle;
endfor;
```

Now we draw the curves. Notice that we used zero widths for the first two characters in order to simplify overlapping (look how fig.

```
beginchar ("a",0,height#,0);
pickup pencircle scaled penw0;   draw p0;
endchar;

beginchar ("b",0,height#,0);
pickup pencircle scaled penw1;   draw p1;
endchar;

beginchar ("c",width#,height#,0);
pickup pencircle scaled penw2;   draw p2;
endchar;
```

Skip this at the first reading

We have made some global definitions which might spoil further work. We therefore undefine the points. It is achieved by assigning `whatever`. It is done here for safety because the examples are extracted from several METAFONT files designed by the author and the global definitions might interfere with something. However, a normal user usually does not need it.

```
for k:=0 upto 60: z[k]=(whatever,whatever); endfor;
```

Scientific graph

Now we make an example of presentation of scientific results. Imagine that we have measured vapour pressures of some chemical species and afterwards we have found the best fit in the form

$$\log p = A - \frac{B}{t + C} \tag{1}$$

where t is temperature in degrees Centigrade and p is pressure in kilopascals. Numerical values of parameters A, B, C are defined later in the METAFONT source.

As you can see, the temperature ranges from 60 ∘C to 90 ∘C and pressure ranges from 80 kPa to 170 kPa. We therefore need some scaling and shift of the origin. A novice might read about `currenttransform` and try to harness it. This, however, has undesirable side-effects and therefore we suggest to avoid it. It is better to use simple linear transform defined with macros.

```
def degC = degCa + degCb enddef;
def kPa = kPaa + kPab enddef;
```

Now let's examine what happens if we write 75degC. This expression expands to 75degCa + degCb. It's clear how the transform works. We must only emphasize that 75degC is not equal to degC*75 because degC is not a variable but a macro.

Now we can start the plot. We specify the dimensions of the character, define the temperature–pressure coordinates of the lower left and upper right corners (METAFONT evaluates degCa, degCb, kPaa, kPab for us) and specify parameters A, B, C and seven experimental points.

You will see special variables w and h. At the time of reading `beginchar` METAFONT assigns width to w, height to h and depth to d. All these variables are expressed in pixels rounded to whole numbers.

```
beginchar("B",100u#,99u#,0);
origin = (50degC,50kPa);  (w,h) = (100degC,200kPa);
A = 3194;  B = 605;  C = 232;
z1 = (60degC,80kPa);  z2 = (65degC,92kPa);  z3 = (70degC,105kPa);
z4 = (75degC,119kPa);  z5 = (80degC,134kPa);
z6 = (85degC,151kPa);  z7 = (90degC,170kPa);
```

It is tedious to type this by hand but it can be prepared by the program which is used for finding the best fit.

In this case the best fit was expressed in a way which can be evaluated with METAFONT. This is not a frequent situation. The easiest way is to tabulate the best fit in many points (do it with your software and make the output suitable for input to METAFONT) and connect them with a crooked line. You will use a similar technique as below. The only difference is that you will define the points but we are calculating them. It is of course possible to draw a curve which is not mathematically defined as the best fit. In such a case you should specify a very small number of points and play with directions and tensions. The next part shows that the index expression may even be a real number.

```
for t := 55 step .3 until 95:
x[t] = t*degC;
y[t] = (mexp(A - 1000/(t+C)*B))*kPa;
endfor;
pickup pencircle scaled 1.5pt;
draw z55 for t:= (55+.3) step .3 until 95: --z[t] endfor;
```

METAFONT has some limitation for calculations. Value 4096 is treated as infinity. Greater values can appear in calculations but they must be less that 32768. Therefore, the values $B = 605000$ would cause arithmetic overflow. Due to it we had to modify equation

We have seen another useful feature of METAFONT. The loop command may even be used in the middle of expression. Here it was used inside the `draw` command.

We also want to see the experimental points. We will draw them as squares.

```
pickup pensquare scaled 4u;
for k:= 1 upto 7: drawdot z[k]; endfor;
```

At last we draw the frame with marks for 75 ∘C, 100 kPa, and 150 kPa.

```
pickup pensquare scaled .7pt;
draw origin--(0,h)--(w,h)--(w,0)--cycle;
pickup pencircle scaled .3pt;
draw (75degC,0)--(75degC,5u);
draw (0,100kPa)--(5u,100kPa);
draw (0,150kPa)--(5u,150kPa);
endchar;
```

Notice that we specified the position of marks in the corresponding units. We could as well use $w/2$ or even $50u$ instead of $75degC$. Such things are, however, too absolute. If you for some reason change the with to $150u\#$, $50u$ will no longer correspond to 75 ∘C. You can change the temperature range to 60 ∘C–120 ∘C and now $w/2$ corresponds to 90 ∘C. It is clear that $75degC$ is invariate under such changes.

Look how figurehorrible but after some practice you will find it easy.

METAFONT has more advanced mechanisms which could be harnessed for transfer of dimensions and coordinates. Some macro packages as `incpic.mf` and `incpic.tex` by Oldřich Ulrych make use of it. But this is for experts (or those who do not care how it works inside). A novice would have hard times to understand it. If you know the mechanism, you cam find your own bugs and you can modify it so that it satisfies your personal needs.

The easiest way seems to be the standard LaTeX' picture environment. To avoid some calculations, we place the origin of the environment into the origin of our graph. All texts are aligned using `\makebox` commands. The dimensions are specified in truemm and truecm. These units remain the same if you change the `\magnification`. It cannot be done in LaTeX but it is used here in case someone would like to incorporate similar concepts into plain TeX.

More complex examples

This section can be too difficult for novices. We show more advanced macro definitions. If you cannot understand it at the first reading, just skip this chapter and return here after you make several own pictures. However, **do not forget to read the important warning later in this document!**

The next part of METAFONT code is best placed at the beginning of the file so that you can fiddle with the parameters. In this example it is placed here in order not to disturb the initial explanation with hard to understand commands.

We start with some parameter definitions. Note that two variables are declared as `pair`.

```
pair tieshift, tiedepth;
smallcorner = 1.5u;  bigcorner = 7.5u;
slope = 3;
tieshift = down scaled 2.5u;
tiedepth = down scaled 4.5u;
tv = 3;  % this is tension for ties
```

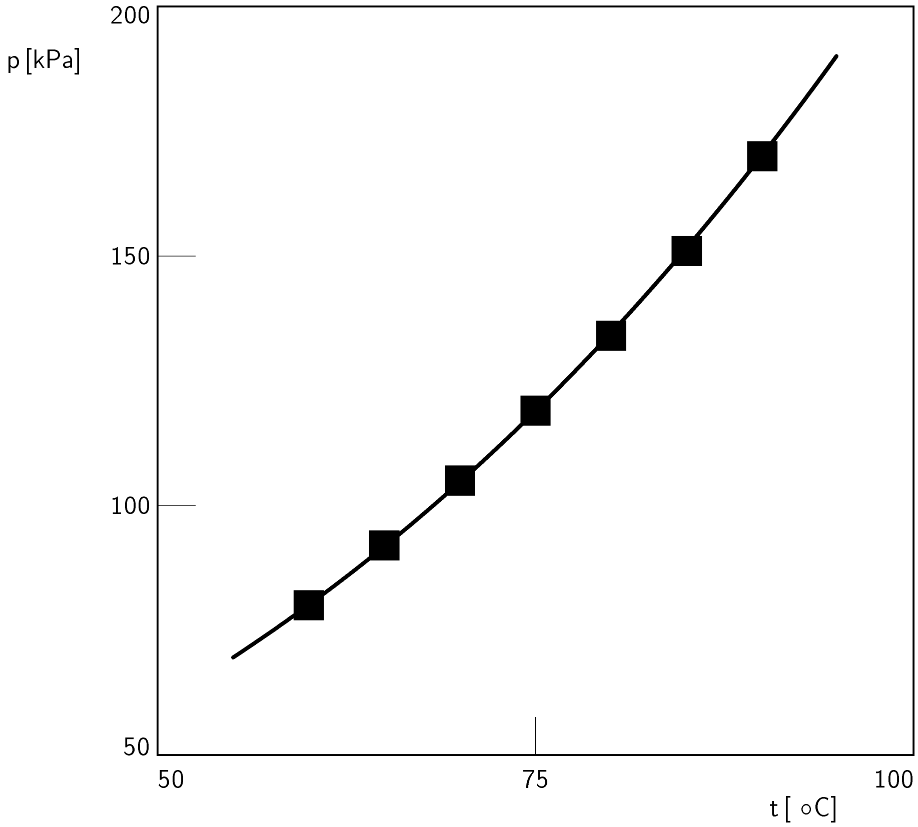

Figure 3: Vapour pressure curve

We have already seen commands for pen selection. They are quite slow. If we pick up the same pen many times, it is faster to store the pen in some variable using `savepen`. We do that with two different pens.[2]

```
pickup pencircle scaled 1pt;
normalpen := savepen;

pickup pencircle scaled .4pt;
penfortie := savepen;
```

[2]It may seem rather useless for two pictures but remember that this example is a small part of a large font.

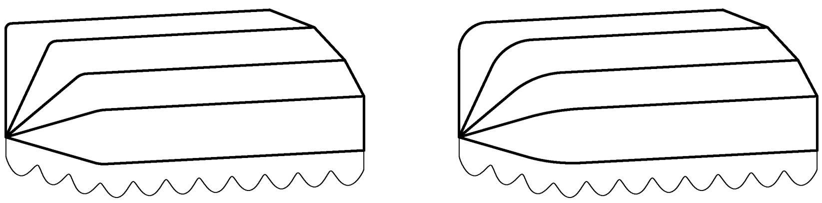

Figure 4: Drawings of marquises

Now we define a macro with parameters. This macro should draw a single segment of a tie. As you can see in fig.should be composed of many segments. Therefore there is no semicolon inside the macro definition.

```
def tiebelowline(expr l, r, t, u) =
(t[l,r] shifted tieshift)..tension tv and 1..
 ((.5[t,u])[l,r] shifted tiedepth)..tension 1 and tv
enddef;
```

We will need two different round corners. There's another place for macro with parameters. It contains cryptic commands. So we write the macro and explain it below.

```
def roundcorner(expr a, b, c, r) =
begingroup save q, w;  pair q, w;
hide(save __p;  path __p;
__p = fullcircle scaled r shifted b;
q := (a--b) intersectionpoint __p;
w := (b--c) intersectionpoint __p; )
a--q{b-a}..{c-b}w--c
endgroup
enddef;
```

This macro should draw a line from a to c where the sharp corner at b is replaced by a part of circle of diameter r. the macro uses its own internal variables. Not to spoil other things in our source, we close the calculations into a group. It is similar to TEX groups but the behaviour is slightly different. We must explicitly `save` the variables. After that METAFONT forgets whatever meaning they might have had and then we can define them. similarly as `tie`, macro `roundcorner` expands to a segment of a longer path. Therefore we must hide the calculations so that METAFONT does not see them when constructing the path. This is by saying `hide` and closing the hidden code into parentheses.

The hidden code starts with saving `__p` and declaring it as a variable of type `path`. It is then defined to be a circle of diameter r with the centre at point b. The next two lines of code calculate the points of intersection of the full circle (path `__p`) with straight lines (`a--b`) and (`b--c`) and assigns them to q and w, respectively. now we can construct the segment. We specify directions st q and w. Again semicolon does not appear here because it should be used as a part of a longer path.

We have said that w contains the width of the character rounded to the whole number of pixels. Now we use it as a variable of type pair. You may wonder why METAFONT does not get confused. The reason is that we did the change inside a group. We saved the old meaning which is automatically restored when METAFONT performs `endgroup`.

We are going to draw two similar marquises. They will differ in one parameter only. Therefore we write another macro. First we define some points. These definitions must be global. We will use variable i for some calculations. We adopt a rule that this variable serves as a loop control variable and is not used for any other purpose. Therefore we need not save it.

We will see a new token `rotated`. This denotes rotation of the endpoint around origin.

Scaling, shift and rotation are transformations. Shift and rotation are not commutative. It means that it is important to apply them in the correct order. In the following macro you can find

```
z[i+10] = right scaled 35u rotated slope shifted z[i];
```

As an exercise change it to[3]

```
z[i+10] = right shifted z[i] scaled 35u rotated slope;
```

and look what it makes with the marquise[4]

```
def Kcxi(expr corner) =
z1 = origin;
z2 = right scaled 13u rotated -16;
z3 = right scaled 13u rotated 16;
z4 = right scaled 13u rotated 40;
z5 = right scaled 14u rotated 65;
z6 = (0,15u);
for i := 2 upto 6:
z[i+10] = right scaled 35u rotated slope shifted z[i];
endfor;
pickup normalpen;
draw roundcorner(z1,z2,z12,corner);
for i := 3 upto 6:
  draw roundcorner(z1,z[i],z[i+10],corner)--z[i+9];
endfor;
pickup penfortie;
draw z1--tiebelowline(z1,z2,0,1/3)..tiebelowline(z1,z2,1/3,2/3)
  ..tiebelowline(z1,z2,2/3,1)..
  for i := 1 upto 8: tiebelowline(z2,z12,(i-1)/8,i/8)..endfor
  z12 shifted tieshift--z12;
enddef;
```

[3]If you happen to corrupt drawing.mf, do not despair. Just erase drawing.aux and run drawing.tex through LaTeX. It will recreate drawing.mf.

[4]Are you really doing the exercise or just reading the text? your own practice will give you much more. Of course you can also try your own pictures.

The forming of the marquises is now easy. We just call the macro with the correct corner. Please notice that those characters have nonzero depths.

```
beginchar("C",50u#,16u#,12u#);
Kcxi(smallcorner);
endchar;

beginchar("D",50u#,16u#,12u#);
Kcxi(bigcorner);
endchar;
```

Important warning

At the end of the METAFONT code we have to place

```
end
```

Semicolon is not required here (but you can use it) because METAFONT ignores everything which might appear after the `end` token. **It is important to place end–of–line character at the last line of code.** If you forget it, METAFONT will award you with a horrible message

```
! METAFONT capacity exceeded, sorry [buffer size=500].
l.132
      end^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?^^?...
If you really absolutely need more capacity,
you can ask a wizard to enlarge me.
```

Conclusion

We have seen simple examples of drawings produced by METAFONT. There are plenty of other commands which we have not discussed here. One of them is `fill` which fills in a cyclic path (try to define one and fill it by saying e.g.

```
fill z1..z2..z3..cycle;
```

where all points were previously defined). If you master METAFONT, you can make lots of tricks.

I said that MFpic inserts an additional step which slows down the progress when you need to fine tune the curves. From this document it may seem that I did not make any improvement. The truth is that in your life you will not use the `mfcode` environment. It is used in this example to ensure that everything is distributed together. Normally the METAFONT source code is written directly and is not created by running TEX.

I wanted to demonstrate that you have to know relatively small number of METAFONT commands in order to draw simple pictures. If you try this, it will encourage you to study METAFONTbook. It is useful although you will probably never design your own letters.

It may happen that METAFONT is extremely cumbersome for some particular case. Then you are free to scan an image using your scanner, modify it with e.g. Correl Draw, overlap it with pictures designed with METAFONT and texts written in TEX, cut and paste it using the functions provided by dvidot or other drivers and polish it by means of *PostScript* features. The only limitation is the availability of different soft- and hardware tools and your own invention.

Zdeněk Wagner
E. Hála Laboratory of Thermodynamics
Academy of Sciences of the Czech Republic
Prague

e–mail: `wagner@csearn.bitnet`, `wagner@earn.cvut.cs`