Introduction to METAFONT

DOUG HENDERSON

Blue Sky Research
534 SW Third Avenue
Portland, Oregon 97204

ABSTRACT

The purpose of this "Introduction to METAFONT" talk is to give a small amount of historical background on what METAFONT is, to introduce a few key concepts and METAFONT commands, and to go over a few more complicated examples and commands. It is beyond the scope of this twenty-minute talk to explain how METAFONT works in detail, but I hope you find METAFONT as interesting as I do, and I hope that I do not verbally wander off on you — at least, not too far.

1. What is METAFONT?

METAFONT is a very powerful tool for producing fonts. Created in 1981 by Prof. Donald E. Knuth, it has undergone quite a few changes to bring it to its current state. Prof. Knuth needed to create the TEX typesetting program/language to be able to create the beautiful math which he was familiar with in his *Art of Computer Programming* books, and METAFONT is the companion program which creates typefaces for TEX to use. TEX can be labeled a markup language, since one embeds control sequences in a document, and TEX processes the file accordingly. METAFONT is similar in that it too has an extremely powerful language, but with METAFONT, the user specifies commands which direct METAFONT to place strokes of an electronic pen on a "digital canvas". We will be exploring some of the basic METAFONT commands to get a better understanding of these concepts.

2. Coordinate System

METAFONT works in the **cartesian coordinate system**. This means that positive coordinates are found above and to the right of the 0,0 point, which is known as the **origin**. Fig. 1 shows a representation of METAFONT's cartesian coordinate system. Most METAFONT characters are drawn in the top right quadrant (A, where x and y are positive), but characters such as a lowercase **g**, **j**, or **y** have **descenders**, which extend below the baseline. B represents the **baseline** of a TEX Font Metrics or `tfm` box. For TEX to be able to use characters that METAFONT creates, it needs to know certain things, such as how wide, high and deep characters are, in order to place one character box next to another. This information is kept in the `tfm` file.

Let's look at a few characters and their `tfm` boxes, to see how they fit in METAFONT's coordinate system. Fig. 2 shows the uppercase letter **W**: (w) indicates the **width** of the `tfm` box, (h) is the **height**, and (d) is the **depth** of the box. Fig. 3 shows another character, the lowercase letter **g**, which has a non-zero depth value, and we see that it has a descender, which goes below the baseline. We can also see some labels inside the character; these are called **control points**.

2.1 Control Points

Control points tell METAFONT where to draw, or, more accurately, where to have the digital pen pass through, leaving a wake of ink. Here is one way to assign a value to control point **1**:[1]

```
x1=10;
y1=25;
```

[1] Semicolons are used to separate METAFONT statements.

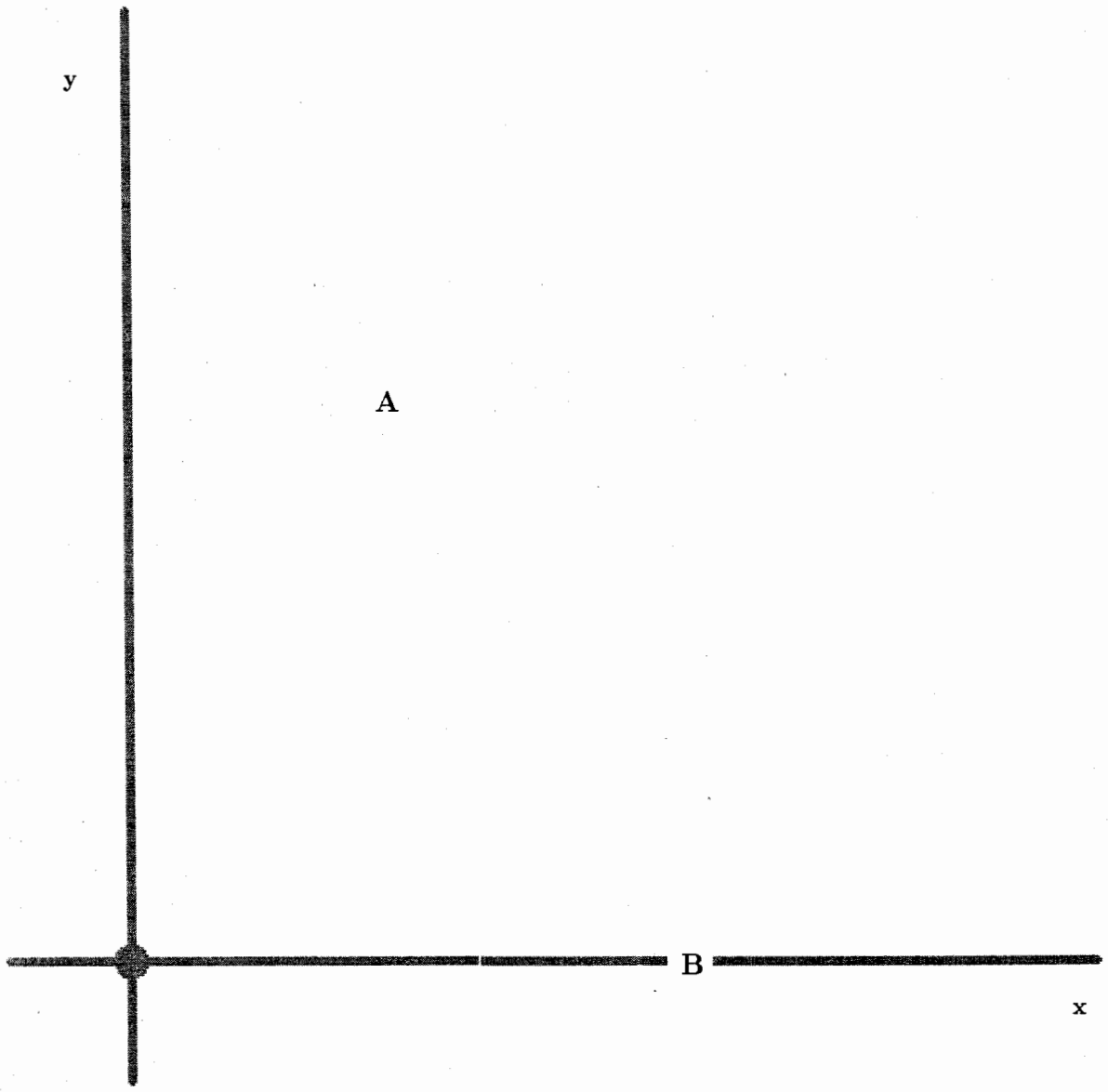


Figure 1: The Cartesian Coordinate System

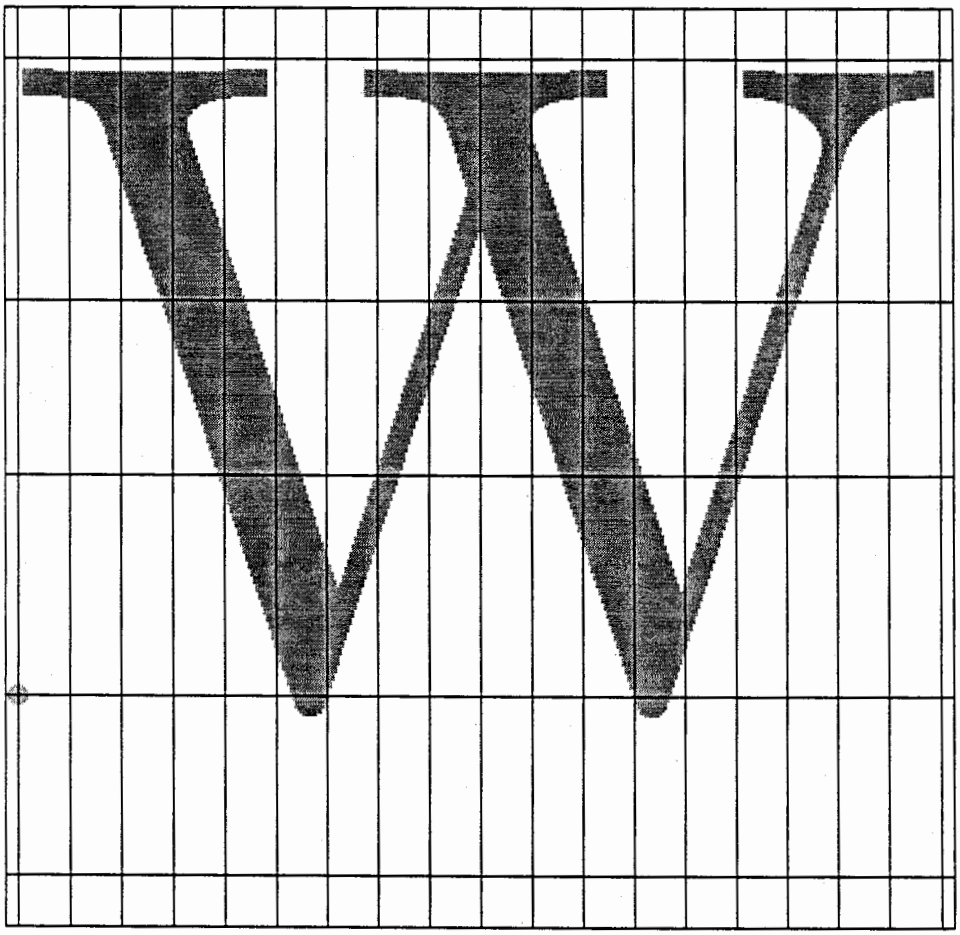


Figure 2: The capital letter **W**

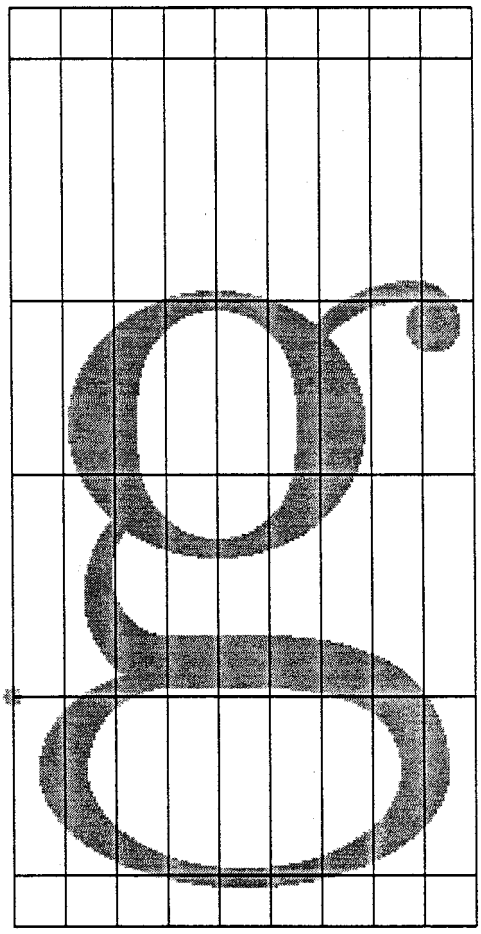


Figure 3: The lowercase letter **g**

We can also assign the same values to the same control point with a single statement like this:

```
(x1,y1)=(10,25);
```

or alternately, a pair of variables can be assigned with a z-point notation where z represents an x-y pair. It is sort of a shorthand method for describing a **coordinate pair**. It looks like this:

```
z1=(10,25);
```

All three statements just shown are equivalent.

Now let's define some more control points and see what happens when we try to draw something with the `draw` command. Here is one way to define some control points:

```
y1=25;
z2=(75,50);
x3=100; x1=y3=y5=10;
z4=(120,-20);
x5=150;
```

Notice that the `x1`, `y3` and `y5` values have all been assigned in one statement as being 10, and that the `z2` and `z4` control points were assigned in a single statement. By combining the x-y assignment and the z assignment methods, we can save quite a bit of typing and also make it clear to METAFONT the relationship between our control points at the same time.

Here is a simple `draw` statement to help us see the path we have defined (after we have started up the demo on the Macintosh, that is):

```
virmf[2] &cm \mode=proof; screenstrokes; input tugcon
draw z1..z2..z3..z4..z5;
```

Figure 4 illustrates the path that results after executing the `draw` statement. So we can get a better feel for what METAFONT is doing, let's look at the individual control points along the curve we just drew (Fig. 5):

```
lose_control(1,2,3,4,5)
```

This macro was one that I created for this conference so it would punch holes in the path of the previous `draw` command, and we could better see how control points are used.

Of course, we don't need to say `draw z1..z2..z3..z4..z5;` with the control points ordered sequentially from **1** to **5**; we can also draw starting and ending at any defined control point. For instance, if we said:

```
clearit;[3]
draw z1..z3..z4..z5..z2;
```

instead of our previous order, we would get the shape shown in (Fig. 6)

After we expose the control points, Fig. 7 shows the results of:

```
lose_control(1,3,4,5,2);
```

We can see by the control points inside our drawn path that the curve starts at point **1**, proceeds down to **3**, then curves nicely around to **4** and **5**, and ends up at point number **2**. METAFONT draws nice curves through these points, and in order to continue smoothly to the next point, it needs to swing out a little ways after passing through a control point. The way that METAFONT makes these pleasing curves is internal; all you need to do is specify the control points to draw through and it does the rest for you. You can change how METAFONT draws curves with special curve modifying commands and we might explore a few of these later. For now, let's look at how METAFONT draws curves.

2.2 Curves

When METAFONT draws a curve, it uses something we can simply call "the four-point method". If we have four control points (Fig. 8):

---

[2] `inimf` is know as the initialization version of METAFONT, `virmf` is the production version.

[3] The `clearit;` statement is one which we use to erase the previous picture that METAFONT was saving for us so we can draw again.

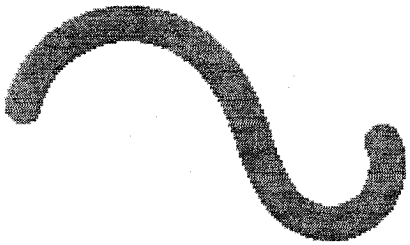


Figure 4: Curve 1 2 3 4 5 with default pen

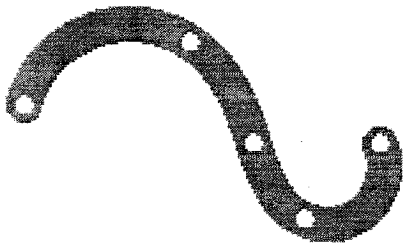


Figure 5: Curve 1 2 3 4 5 with exposed control points

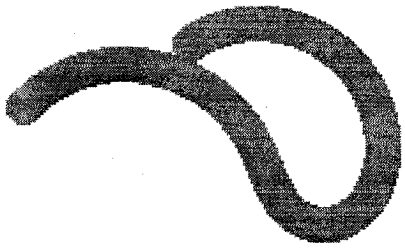


Figure 6: Curve 1 3 4 5 2 with default pen

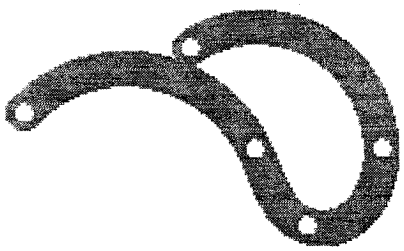


Figure 7: Curve 1 3 4 5 2 with exposed control points

Figs. 4–7: Curves and Control Points

```
z1=(35,100); z2=(60,10); z3=(200,10); z4=(225,75);
```

the curve that METAFONT would draw is found by repeated mid-point calculations, as in Fig. 9.

A more technical name for the curve defined by METAFONT is a Bézier cubic. What METAFONT does for us is take the original control points we supply and add other control points of its own, as we see in Fig. 10. Then it refines the curve between the "scaffolding", as Knuth calls it, until the curve is left on the innermost path between midpoints, which is what we see in Fig. 11. METAFONT then discards the scaffolding and draws a nice curve which is inside the scaffolding.

These are the basics for drawing curves, using this four-point refinement method. There are also other commands which affect how the scaffolding is built. For instance, there are commands which can create more tension in the curve, such as in Fig. 12, or more curling of curves at the endpoints (Fig. 13).

The degree to which you can manipulate METAFONT curves is really quite astounding. Unfortunately, there is not enough time to go into all the ways to generate different curves with METAFONT.

2.3 Pens

Another interesting concept is that of a METAFONT pen. A good way to view the sizes and strokes we use to draw with METAFONT is to think of them as being produced by nibs of different pens (because, in fact, they are). Until now, we have only used one pen type for our examples and since we didn't specify, METAFONT provided us with a default pen. Let's look at some different pen types and how to use them.

Before you start drawing with a pen, you generally have to pick it up first, and here is how we tell METAFONT to do just that:

```
pickup pencircle;
```

In addition to a circular nibbed pen, there are a few other pen types that METAFONT knows about (through definition in the plain base file). They are:

```
pensquare
penspeck
penrazor
```

`penspeck` and `penrazor` are special-purpose in nature; `penspeck` is used in the `drawdot macro`, and `penrazor`, as the name implies, is a razor-thin pen (one pixel). `pencircle` and `pensquare` perform mostly as you would expect of pens with such names. Let's look at how we can specify different pen nibs via some examples:

```
% clear drawing board, but not control points
clearit;
% pickup a pen to draw with
pickup pencircle;
% and draw !
draw z1..z2..z3..z4..z5;
```

As we see, this is the pen we used before (the default pen). Let's look at a few ways to "build" some pens for METAFONT to use. One way to change our pen is by scaling it to the size desired. There are three scaling commands: `scaled`, `xscaled`, and `yscaled`. Here is a command which scales a pen to nearly one tenth point size:

```
clearit;
pickup pencircle scaled .1pt;
draw z1..z2..z3..z4..z5;
```

Notice the size difference from the last pen we used. This pen (Fig. 14a) is much smaller than our default in Fig. 4, which was approximately .4pt.

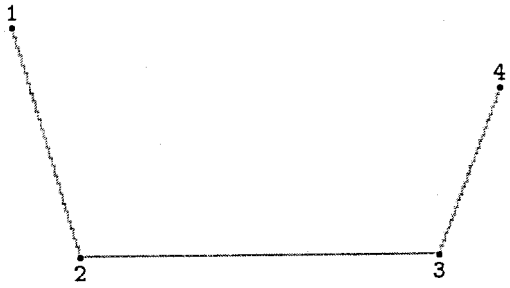


Figure 8: Four-point method 1 2 3 4 draw

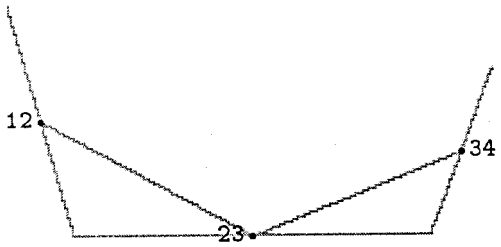


Figure 9: Four-point method 12 23 34 draw

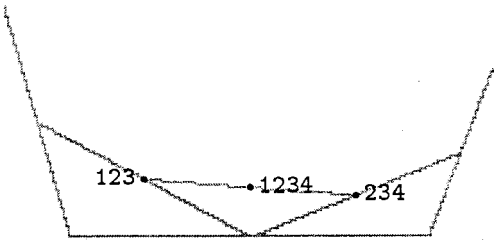


Figure 10: Four-point method 123 1234 234 draw

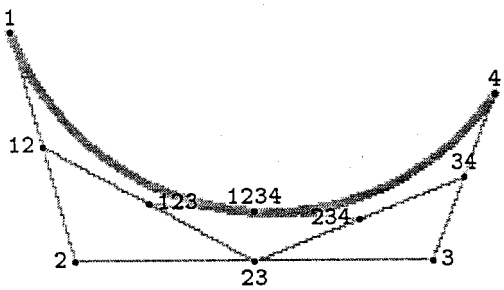


Figure 11: Four-point method 1 1234 4 draw

Figure 12: Four-point method 1 tension 2 1234 tension 2 4 draw

Figure 13: Four-point method 1 curl infinity 1234 curl infinity 4 draw

Figs. 8–13: Four Point Method

Figure 14a: Pen = .1pt

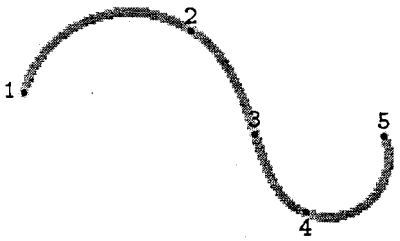


Figure 14b: Curve with pen = .1pt

Figure 15a: Pen = xcale=.6pt, yscale=.2pt

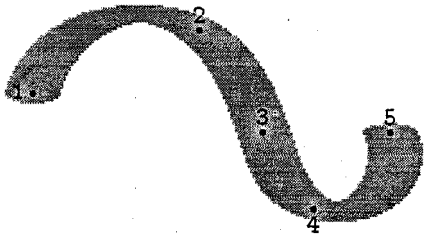


Figure 15b: Curve with pen = xscale=.6pt, yscale=.2pt

Figure 16a: Pen = xscale.2pt, yscale.6pt

Figure 17a: Pen = xscale.2pt, yscale.6pt, rotate 32 degrees

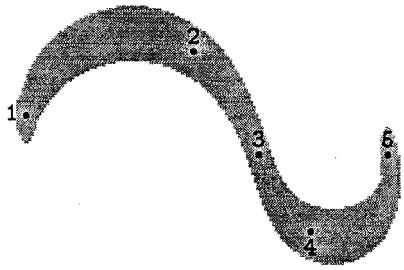


Figure 16b: Curve with pen = xscale=.2pt, yscale=.6pt

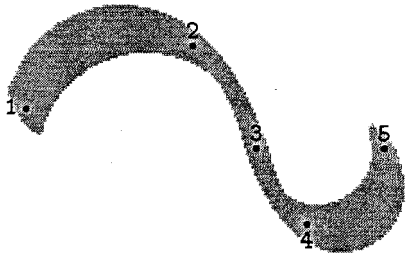


Figure 17b: Curve with pen = xscale.2pt, yscale.6pt, rotate 32 degrees

Figs. 14–17: Pen Building

The following command introduces the x and y scaling operation:[4]

```
clearit;
pickup pencircle xscaled .6pt yscaled .2pt;
drawem;
```

In Fig. 15a, we can see the pen nib is wide and short, since the xscaling is greater than the yscaling. If we switch the x and y scaling, like this:

```
clearit;
pickup pencircle xscaled .2pt yscaled .6pt;
drawem;
```

we get the results shown in Fig. 16a, where the nib is thin and tall, since the yscaling is greater than the xscaling.

Another parameter of control that one has for pen manipulation, aside from scaling, is **rotation**. Here is a sample (Fig. 17a) that has the pen rotated 32 degrees with the same xscaling and yscaling as the previous example (`x=.2; y=.6`):

```
clearit;
pickup pencircle xscaled .2pt yscaled .6pt rotated 32;
drawem;
```

This last pen seems to emulate a calligraphic pen, with the rotation acting as the angle of a pen being held by a hand.

This concludes the section on METAFONT commands. Now I will attempt to give a brief history of METAFONT, and then show a little of the power behind METAFONT.

3. METAFONT — Evolution of a Program

Originally, METAFONT had only 28 parameters which described the small pieces which make up a complete character. After working with Hermann Zapf, Mathew Carter, Charles Bigelow and Kris Holmes (receiving much feedback from them all in 1981), Knuth worked very hard at bettering his original typefaces. Then, in April 1982, Richard Southall came to Stanford and helped make extensive changes to the Computer Modern programs (especially the sans serif letters). This resulted in the refinement of the METAFONT language and brought the number of parameters to 45. Although small refinements occasionally surface in the Computer Modern typefaces, they remain today steady and stable with the total number of parameters at 62 as there have been since 1985 (see Appendix A for a list of the parameters for `cmr10`).

So one of the key ideas behind a METAFONT is that there are a large number of parameters to describe what a character looks like. By varying these parameters, we can see how different typefaces are created. Let's look at some differences in parameters by viewing the result of a test file named `6test`, which uses six different sets of parameters to create six variations of the same character.

```
virmf & \mode=proof; mag=.33; screenchars; input 6test
```

As we can see in Fig. 18, there are six different characters that have been generated on the screen. They are: `cmr10` or Computer Modern Roman (top left), `cmss10` or Computer Modern Sans Serif (top middle), `cmtt10` or Computer Modern Typewriter (top right), `cmb10` or Computer Modern Roman Bold (bottom left), `cmbx10` or Computer Modern Roman Bold Extended (bottom middle), and `cmti10` or Computer Modern Text Italic (bottom right).

4. A Word About Computer Modern

The Computer Modern typefaces, 75 in all, comprise the effort put forth by Knuth to create the spirit of the typeface Monotype Modern 8A, which has traditionally been used to print textbooks of all sorts, including Knuth's first two volumes of *The Art of Computer Programming*. If one doesn't like Computer Modern, I believe it is because one doesn't like Monotype Modern 8A, not because Knuth made a poor rendition of same.

---

[4] I was lazy and didn't want to make too many typos in my demo, so I created the macro **drawem**, which draws from control points 1 to 2 to 3 to 4 to 5.

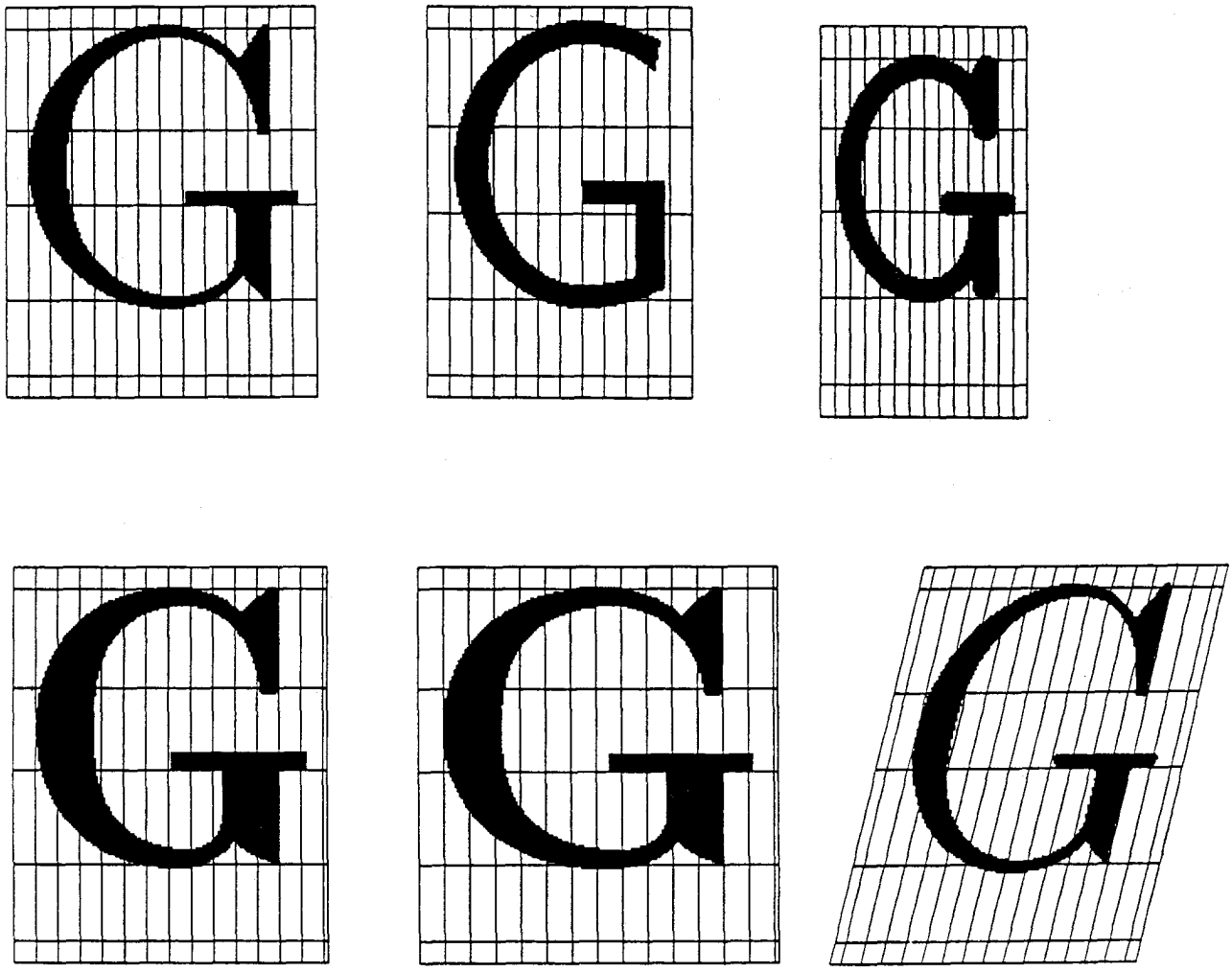


Figure 18: Results of the `6test` file

An extreme example of this misunderstanding that I have encountered, occurred when someone waved an Epson dot matrix folio in my face, and exclaimed "this is ugly". Well, looking at it, I had to agree. And when this person further claimed that the sad looking characters on the page didn't look anything like Times Roman (obviously what he expected) I also had to agree. I think this misunderstanding is common. People often view Computer Modern and say to themselves "Why didn't Knuth typeset his first volumes in Garamond or Palatino?", just wishing that Computer Modern was actually one of those, or perhaps Times Roman (these fonts happen to be in vogue now). Well, he didn't and they are not. They should not be compared in this apples-are-better-than-oranges way. Besides, Garamond and Palatino fonts are proprietary fonts; source files would certainly not be available — as they are for the CM fonts — for users to modify and alter and re-shape.

The challenge which lies ahead is for brave souls to create new typefaces, or adapt classic typefaces to satisfy the TUG community. We can all start with some understanding of typography, and the basics of METAFONT, and go from there.

Any volunteers?

Bibliography

Knuth, Donald, E. *The Art of Computer Programming*, vols. 1–3 Reading Mass.: Addison Wesley, 1968, 1973

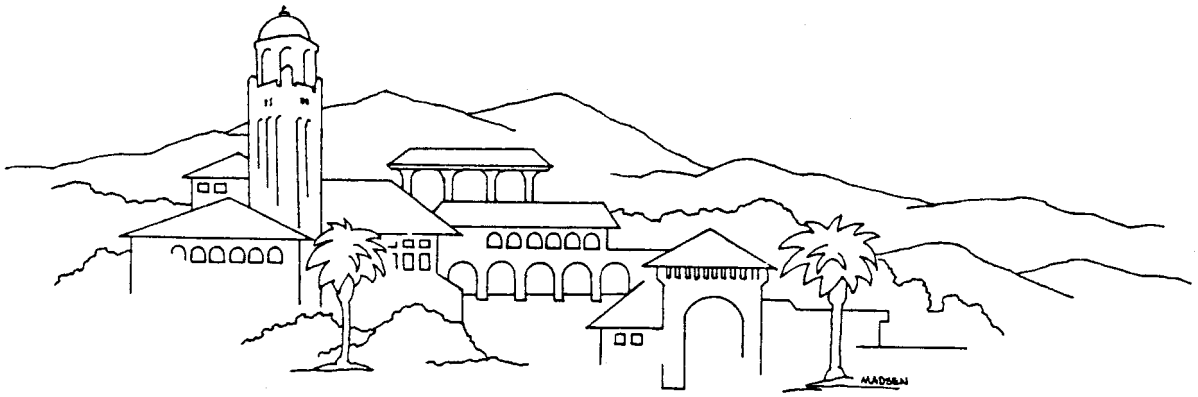
Knuth, Donald, E. *The* METAFONT*book Computers and Typesetting,* Vol. C. Reading, Mass.: Addison Wesley, 1986.

Knuth, Donald E. METAFONT*: The Program. Computers and Typesetting*, Vol. D. Reading, Mass.: Addison-Wesley, 1986.

Knuth, Donald E. *Computer Modern Typefaces. Computers and Typesetting*, Vol. E. Reading, Mass.: Addison-Wesley, 1986.

Appendix A

```
% This is CMR10.MF in text format, as of Mar 31, 1986.
% Computer Modern Roman 10 point
if unknown cmbase:  input cmbase fi
font_identifier:="CMR"; font_size 10pt#;
u#:=20/36pt#;                    % unit width
width_adj#:=0pt#;                  % width adjustment for certain characters
serif_fit#:=0pt#;                  % extra sidebar near lowercase serifs
cap_serif_fit#:=5/36pt#;                 % extra sidebar near uppercase serifs
letter_fit#:=0pt#;                 % extra space added to all sidebars
body_height#:=270/36pt#;                 % height of tallest characters
asc_height#:=250/36pt#;                  % height of lowercase ascenders
cap_height#:=246/36pt#;                  % height of caps
fig_height#:=232/36pt#;                  % height of numerals
x_height#:=155/36pt#;                  % height of lowercase without ascenders
math_axis#:=90/36pt#;                  % axis of symmetry for math symbols
bar_height#:=87/36pt#;                   % height of crossbar in lowercase e
comma_depth#:=70/36pt#;                  % depth of comma below baseline
desc_depth#:=70/36pt#;                   % depth of lowercase descenders
crisp#:=0pt#;                  % diameter of serif corners
tiny#:=8/36pt#;                  % diameter of rounded corners
fine#:=7/36pt#;                  % diameter of sharply rounded corners
thin_join#:=7/36pt#;                   % width of extrafine details
hair#:=9/36pt#;                  % lowercase hairline breadth
stem#:=25/36pt#;                   % lowercase stem breadth
curve#:=30/36pt#;                    % lowercase curve breadth
ess#:=27/36pt#;                  % breadth in middle of lowercase s
flare#:=33/36pt#;                  % diameter of bulbs or breadth of terminals
dot_size#:=38/36pt#;                   % diameter of dots
cap_hair#:=11/36pt#;                   % uppercase hairline breadth
cap_stem#:=32/36pt#;                   % uppercase stem breadth
cap_curve#:=37/36pt#;                   % uppercase curve breadth
cap_ess#:=35/36pt#;                  % breadth in middle of uppercase s
rule_thickness#:=.4pt#;                    % thickness of lines in math symbols
dish#:=1/36pt#;                  % amount erased at top or bottom of serifs
bracket#:=20/36pt#;                    % vertical distance from serif base to tangent
jut#:=28/36pt#;                  % protrusion of lowercase serifs
cap_jut#:=37/36pt#;                    % protrusion of uppercase serifs
beak_jut#:=10/36pt#;                   % horizontal protrusion of beak serifs
beak#:=70/36pt#;                   % vertical protrusion of beak serifs
vair#:=8/36pt#;                  % vertical diameter of hairlines
notch_cut#:=10pt#;                   % maximum breadth above or below notches
bar#:=11/36pt#;                  % lowercase bar thickness
slab#:=11/36pt#;                   % serif and arm thickness
cap_bar#:=11/36pt#;                    % uppercase bar thickness
cap_band#:=11/36pt#;                   % uppercase thickness above/below lobes
cap_notch_cut#:=10pt#;                   % max breadth above/below uppercase notches
serif_drop#:=4/36pt#;                  % vertical drop of sloped serifs
stem_corr#:=1/36pt#;                   % for small refinements of stem breadth
vair_corr#:=1/36pt#;                   % for small refinements of hairline height
apex_corr#:=0pt#;                  % extra width at diagonal junctions
o#:=8/36pt#;                   % amount of overshoot for curves
apex_o#:=8/36pt#;                    % amount of overshoot for diagonal junctions
slant:=0;                % tilt ratio (delta x/delta y)
fudge:=1;                % factor applied to weights of heavy characters
math_spread:=0;                  % extra openness of math symbols
superness:=1/sqrt2;                    % parameter for superellipses
superpull:=1/6;                  % extra openness inside bowls
beak_darkness:=11/30;                    % fraction of triangle inside beak serifs
ligs:=2;                 % level of ligatures to be included
square_dots:=false;                    % should dots be square?
hefty:=false;                  % should we try hard not to be overweight?
serifs:=true;                  % should serifs and bulbs be attached?
monospace:=false;                    % should all characters have the same width?
variant_g:=false;                  % should an italic-style g be used?
low_asterisk:=false;                     % should the asterisk be centered at the axis?
math_fitting:=false;                     % should math-mode spacing be used?
generate roman                 % switch to the driver file
```



TEX Users Group Meeting and Short Course

Stanford University, July 25–30, 1982