Some METAFONT Techniques

Yannis Haralambous*

October 6, 1994

Abstract

This paper presents a few ideas on how to solve certain geometrical problems arising very often in character design, not directly solvable by METAFONTs `plain` macros. The first part of the paper presents two geometrical problems: the *"k problem"* and the *"x problem"*, their solutions using dichotomy, and a different solution using path intersections. The latter was proposed earlier on the net by the author; although geometrically correct, it does *not* work in real-world METAFONT practice: a nice example of METAFONT code . . . to avoid.

The second part of the paper presents two simple macro for drawing "loose" Bézier curves; in a sense, the opposite of the `tension` operator. Finally, the third part solves a problem stated by Alan Hoenig: how to extract text and data out of a METAFONT run, without using the log file. This is done in a straightforward manner by running a Flex-generated preprocessor over the GF file: the Flex code for this utility is given in appendix B.

1 Two Geometrical Problems, Solved by Iterated Calculations

1.1 Description

Suppose you want to design a character 'K', as on the left part of fig. 1. The character should fit inside a box of width w and height h, and should consist of three strokes: the vertical stroke $z_0 - -z_{0'}$, and the two oblique strokes $z_1 - -z_2$ and $z_{1'} - -z_{2'}$. Only constraint: the point $z_{1l} = z_{1'r}$ should be fixed (for example, its coordinates can be $(0, \frac{h}{2})$). So, here is the problem:

Find a stroke $z_1 - -z_2$ *with fixed* z_{1l}, y_{2l}, x_{2r}.

This problem is not trivial, because METAFONT cannot compute pen positions without knowing in advance the angle of the pen (this stands both for defining a new pen with command *pickup pen* as for defining a simulated pen with command *penpos*). Because it arises when designing the letter 'K', we will call this problem, the *"k problem"*.

The next problem is encountered when designing an 'X': as on the right part of fig. 1, suppose you want to draw this letter. Once again it should fit inside the box, and should consist of two strokes. To keep the same notation as in the previous case, we have only given names to the pen positions concerning the upper right part of the letter. In this case the constraints are: z_1 is fixed (and not z_{1l} as in the previous case), as well as y_{2l} and x_{2r}. Here again is the problem, which we call the *"x problem"*:

Find a stroke $z_1 - -z_2$ *with fixed* z_1, y_{2l}, x_{2r}.

Well understood, in both cases the direction of the stroke must be perpendicular to the angle of the pen: all strokes must keep the same width.

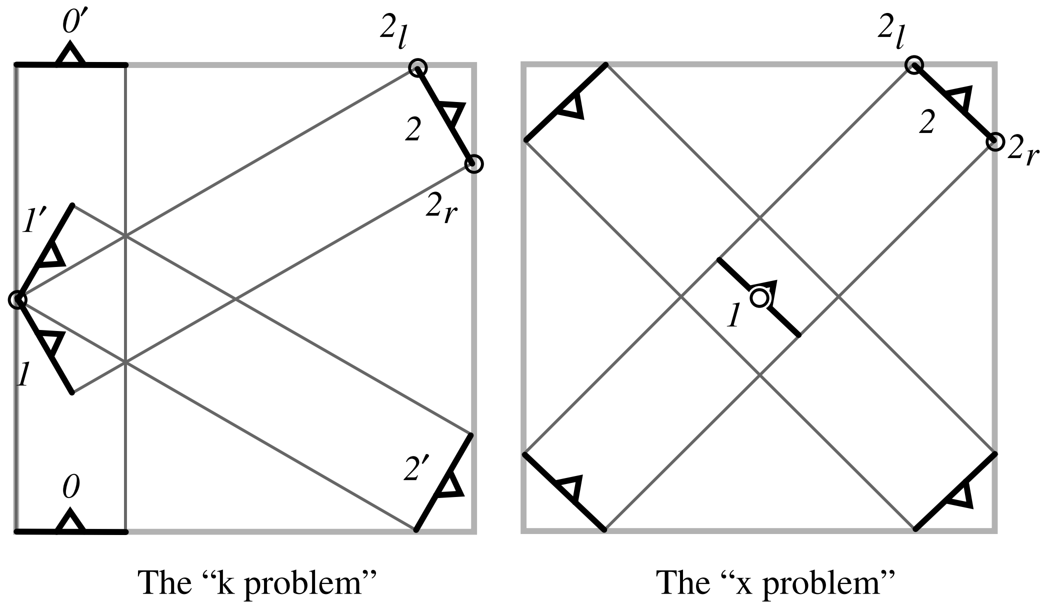

Figure 1: The two problems.

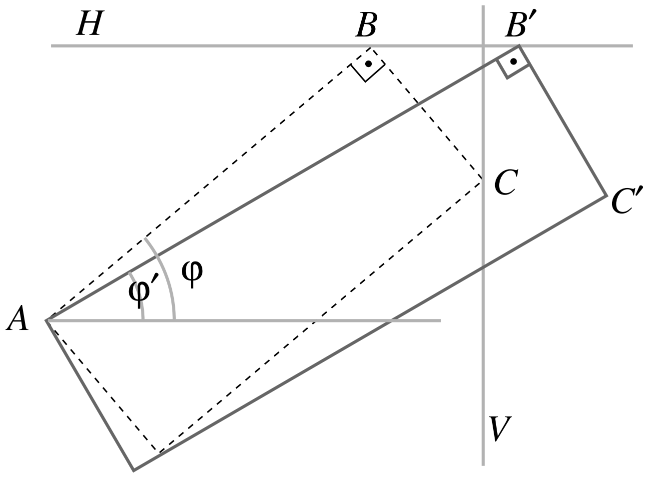

Figure 2: A closer look to the *“k problem”*.

1.2 The Solutions (Which Work)

Let's start with the *"k problem"*. On fig. 2, the reader has a closer look of the situation. Point A is fixed, point B must lie on a fixed horizontal line H, and C on a fixed vertical line V. The angle $\widehat{ABC}$ must stay orthogonal. Also the length $\overline{BC}$ of the vector BC is fixed. METAFONT cannot compute the angle ϕ directly, so that it fits to these constraints. But once we have chosen a point B', METAFONT can calculate the corresponding C' so that $AB' \perp B'C'$ and $\overline{B'C'} = \overline{BC}$.

So let's choose a random point $B' \in H$, and find the corresponding C'. If C' lies on the right of V then we know that we should move B' more to the left, if C' lies on the left of V then B' should be moved more to the right. We will modify the position of B' by a certain step and start again. This procedure will be iterated until we get close enough to V. The step will be halved every time: because of the well-known equality $\sum_{i\geq 1} \frac{1}{2^i} = 1$ we are sure that this process of iterations will converge to the correct result. One may argue that the result will always be approximative; this is true in mathematics but a useless remark in computer calculations, since all values are approximated anyway. Once we have a sufficient precision we stop; the sufficient precision depends on the implementation and the resolution of our character.

This process is called *dichotomy* (from δίχα = in two pieces, and τέμνω = to cut), and is usually one of the first exercices in most programming languages.

Here is the code:

```
def solve_k_problem(suffix $, $$, $$$)(expr pen_width, first_try) =
pair z_zero; z_zero = z$;
numeric x_one, y_one, x_two;
 y_one = y$$; x_one = first_try; x_two = x$$$;
numeric theta, n; n := 1;
forever:
        clearxy; z$ = z_zero; z$$ = (x_one, y_one);
        theta := angle(z$$ - z$); pos$$.1(pen_width, theta - 90);
        if y$ > y$$:
          z$$.1r
      else:
          z$$.1l
      fi = z$$;
   x_one := x_one
   if x$$.1r > x_two: -
   else: +
   fi abs(first_try - x$)/(2 ** n);
   exitif((abs(x$$.1r - x_two) < 0.1) or (n > 13)); n := n + 1;
endfor
 pos$$$(pwidth, theta - 90); z$$$r = z$$.1r;
enddef;
```

This procedure expects that you feed it with: (a) the suffixes of points z_{1l}, z_{2l} and z_{2r}, (b) the width of the pen, (c) a hint on the first choice for x_{2l}. It works usually fine when your hint is simply the x-coordinate of z_{2r}. Theoretically it can go wrong if the x-projection of $z_{1l} - -z_{1r}$ is bigger than he first step of the iteration process. But this can hardly ever happen.

The iteration is stopped when (a) either the distance of z_{2r} to V is less than a tenth of a pixel, or (b) if $n = 14$, because the next step would produce a denominator $2^{15} = 32{,}768$, which is too big for (usual) METAFONT. Experience shows that with 8 steps one is usually done—again, all depends on the resolution.

Let's consider the second problem now. As the reader can see in fig. **??**, only the position of point A differs. Nevertheless, this makes a big difference for METAFONT: in the previous problem, once we had

*Centre d'Études et de Recherche sur le Traitement Automatique des Langues, Institut National des Langues et Civilisations Orientales, Paris. Private address: 187, rue Nationale, 59800 Lille, France. Fax: (33) 20.40.28.64, ISDN/Numéris: (33) 20.15.81.77, E-mail: Yannis.Haralambous@univ-lille1.fr

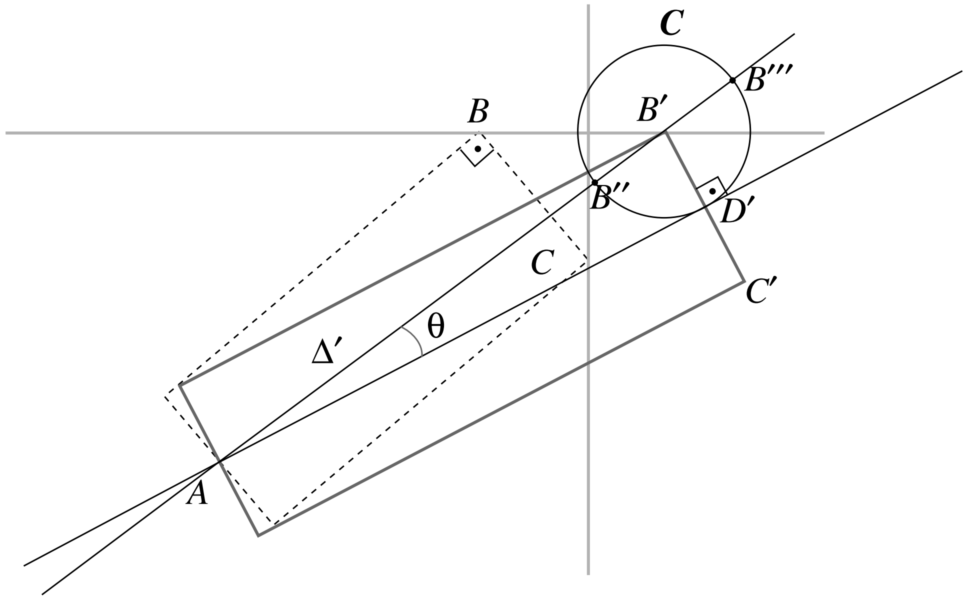

Figure 3: A closer look to the *"x problem"*.

chosen B' we could immediately calculate the position of C'. This is not the case here: all we know is that if D' is the middle of $B'C'$, then $AD' \perp B'C'$. So we need a different technique already to calculate the location of C' for exach step of the iterating process. This will be done again by iteration.

But first let's consider two geometrical facts which we are going to use.

Fact 1. *Let $\mathcal{C}$ be a circle, centered at O, and P a point outside the cercle. For every line Δ, going through P and intersecting the circle at points X and X', the product of lengths $\overline{PX} \cdot \overline{PX'}$ is constant. In particular, if PT is a tangent to the circle going through P, then $\overline{PT}^2 = \overline{PX} \cdot \overline{PX'}$.*

As the reader can see on fig. **??**, it follows from the previous fact that if Δ' is the line going through P and O, and intersecting the circle at Y and Y', then $\overline{PT} = \sqrt{\overline{PY} \cdot \overline{PY'}}$.

The second fact is even more trivial:

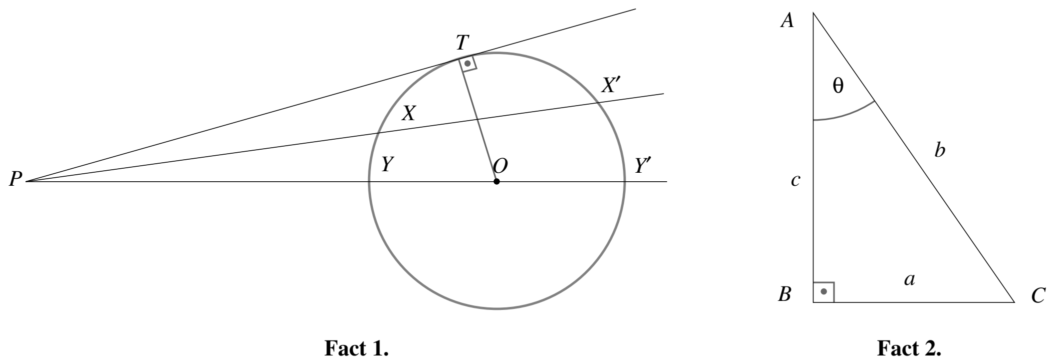

Figure 4: Two facts from elementary Euclidean geometry.

Fact 2. *Let ABC be an orthogonal triangle; the right angle shall be $\widehat{ABC}$; let's call $\widehat{CAB} = \theta$, and a, b, c the lengths of faces opposite to points A, B, C. Then $\theta = \arccos(\frac{c}{b})$.*

Let's return to our problem (see fig. **??**). D' is on a cercle $\mathcal{C}$ centered at B', of radius $\frac{1}{2}\overline{BC}$ (half the width of the stroke, since $\overline{B'D'} = \overline{D'C'}$). Also we know that $AD' \perp B'C' \Rightarrow AD' \perp B'C' \Rightarrow AD'$ is tangent to cercle $\mathcal{C}$. Let's draw the line Δ, going through points A and B'. It will intersect $\mathcal{C}$ at points B'' and B'''. From **Fact 1.** we know that $\overline{AB''} \cdot \overline{AB'''} = \overline{AD'}^2$. So we have not yet D' itself, but the length of AD'.

Let's apply now **Fact 2** to the orthogonal triangle $AD'B'$. We obtain: $\theta = \arccos(\overline{AD'}/\overline{AB'})$. Once we have the angle and the length of AD', we have point D', and we are done for this step of the iterating process. The remaining of the solution is similar to the one of the *"k problem"*: we are moving around B' until C' is close enough to line V.

Let's try to implement this solution to METAFONT. **Fact 1** can be easily implemented: of course one should avoid multiplying two lengths (because of a possible overflow error), but there should be no problem if we take the square root of each length first (for purists: lengths are always positive!). So instead of $\overline{AD'}^2 = \overline{AB''} \cdot \overline{AB'''} =$ we will formulate the equation as $\overline{AD'} = \sqrt{\overline{AB''}} \cdot \sqrt{\overline{AB'''}}$.

The **Fact 2** is a little harder to implement. As a matter of fact, the reader may have noticed that although METAFONT provides exponential *and* logarithmic functions, there are no *inverse trigonometric* functions. What should be do? unfortunately, METAFONT offers no complex calculus so that formulas such as $\cos(x) = \frac{e^{xi}+e^{-xi}}{2}$ could be applied; power series can not be used either because our candidates for angles are not necessarily in a neighborhoud of 0; using an external program to make this calculations would be highly unorthodox. Let's use dichotomy once again!

Here is the code for a *arccosd* procedure in METAFONT:

```
def arccosd(expr ttt) =
if ttt > 1:
    message("error: arccosd argument > 1!!??"); stop;
else: numeric a_;
    numeric test, nnn, prev; test := 45; nnn := 1;
    prev := cosd( test);
    forever:
        nnn := nnn + 1;
        if cosd( test) < ttt:
            test := test − (90/(2 ** nnn))
        else:
            test := test + (90/(2 ** nnn))
        fi;
        exitif((abs( test − prev) < 0.01) or ( nnn > 14));
        prev := test;
    endfor
fi a_ := test; enddef;
```

This procedure requires a number *ttt* $\in]0, 1[$ as argument. It stores the result in the numeric variable *a_*. The first try is always $\frac{\pi}{4}$. Since we want to solve a specific problem (the *"x problem"*), the author has considered *arccosd* only for the first quadrant: solutions will always be in the range $]0, \frac{\pi}{2}[$. One can easily modify the code to work in different ranges[1].

And now the solution of the *"x problem"*:

```
def solve_x_problem(suffix $, $$, $$$)(expr pen_width, first_try) =
pair z_zero; z_zero = z$;
numeric x_one, y_one, x_two;
```

[1] It would be nice to modify the code in order to get complex results for *ttt* $\in]-\infty, 0[\cup]1, \infty[$ but the author can hardly see the utility of these for METAFONT...

```
 y_one = y_{$$};  x_one = first_try;  x_two = x_{$$$};
numeric theta, n, phi, tangent_length; n := 1;
forever:
    clearxy; z_$ = z_zero; z_{$$} = (x_one, y_one);
    theta := angle(z_{$$} − z_$);
    pos_{$$}(pen_width, theta);
    tangent_length := sqrt(length(z_{$$l} − z_$)) ∗ sqrt(length(z_{$$r} − z_$));
    arccosd(tangent_length/ length(z_{$$} − z_$));  phi := theta − a_;
    pos_$(2 tangent_length, phi); z_{$$.1r} = z_{$r} + (z_{$r} − z_{$$}); z_{$$.1l} = z_{$$};
    x_one := x_one
        if x_{$$.1r} > x_two: −
      else: +
      fi
      abs(first_try − x_$)/(2 ∗∗ n);
    exitif((abs(x_{$$.1r} − x_two) < 0.1) or (n > 13)); n := n + 1;
endfor
z_{$$$l} = z_{$$.1l}; z_{$$$r} = z_{$$.1r}; z_{$$$} = 0.5[z_{$$$l}, z_{$$$r}];
enddef;
```

A word of explanation is perhaps necessary, concerning pen position \$\$. This is a quick way to obtain the intersection of line Δ' (fig. **??**) and circle $\mathcal{C}$: the pen \$\$ lies on Δ' and points $\$\$_r$ and $\$\$_l$ are at the right distance from point \$\$. As we will say in the following section, through this method we obtain a (sufficiently) accurate result.

The same idea has been used to explicitly define point \$\$.1: by taking pen position \$, the right edge of the pen is on point D' of the figure.

The entries for this procedure are the same as for *solve_k_problem*. Again the problem is solved in the specific case of the right and upper part of letter 'X'; for the other possible cases, one should either change the code (straightforward but tedious), or use symmetry arguments.

1.3 A Solution Which Doesn't Work, and Why

The author must confess that the first time he had to solve the *"x problem"*, was during a METAFONT tutorial at the Royal Holloway College (UK) [this shows how badly the tutorial was prepared. . . mea culpa]. On the spot he could find no solution, and on the ship across the Channel[2] he found the following one:

Let $\mathcal{C}$ be a circle, centered at B' and of radius $\frac{1}{2}\overline{BC}$, as in fig. **??**. The intersection of Δ' and $\mathcal{C}$ gives us points B'' and B'''. From **Fact 1**, we obtain the length of AD'. Now, D' is at known distance from A, and upon circle $\mathcal{C}$. Take a circle $\mathcal{C}'$, centered at point A and of radius $\overline{AD'}$. The (right) intersection of cercles $\mathcal{C}$ and $\mathcal{C}'$ is the wanted point D'.

This method is mathematically correct—but if you try it out you will get extremely bad results. The problem is that when we define a **path**, METAFONT does not consider it as an abstract curve, but as a set of pixels. When we ask for the intersection of two paths, we obtain the *pixel* which is the closest to the (theoretical) intersection of the paths. In the solution sketched above, the intersection of the two circles is taken as an abstract point, and its coordinates are used for calculations. The result is of course completely deformed.

2 Loosening Bézier curves

Bézier curves are quite beautiful, and METAFONT allows us to obtain them even out of only *partial* information: for example, one can ask for "a curve leaving point A following a vertical direction and arriving at point B following an horizontal direction". There is an infinite number of Bézier curves with

[2] In future, people taking the tunnel will certainly write better METAFONT code.

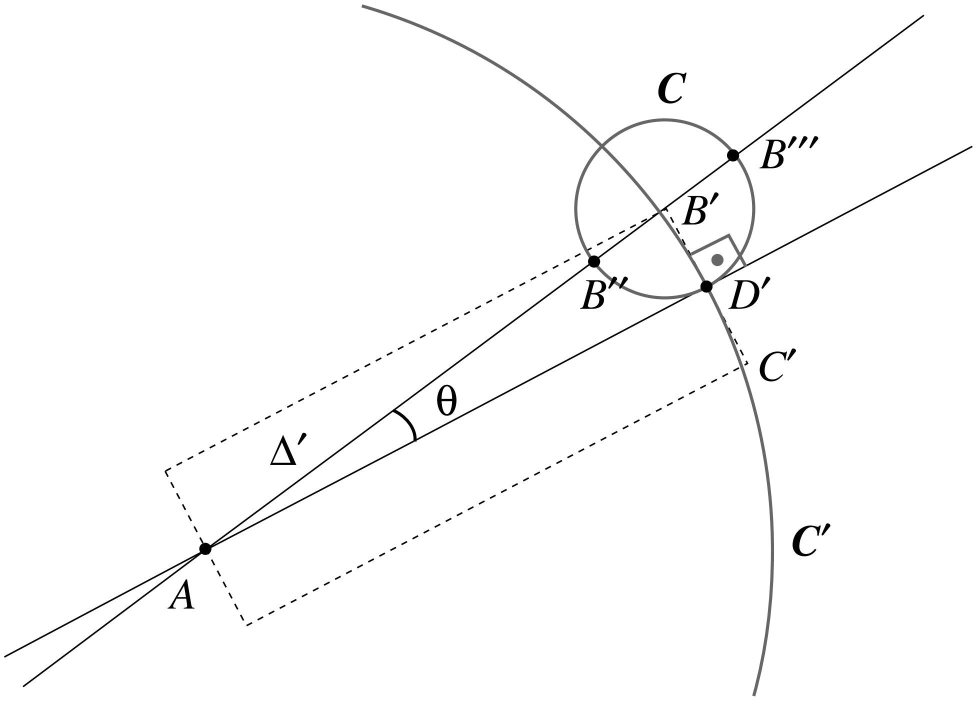

Figure 5: A solution of the *"x problem"* which doesn't work on METAFONT.

exactly these features; METAFONT will choose one of them, out of hard-wired criteria. Most of the time, METAFONT's choice is exactly what you need; but it may also happen that you want to keep some other curve in the same set. There are two operators allowing us to do this:

1. `tension`, which allows us to get tense curves;
2. `controls`, by which we can explicitly determine the control points of our Bézier curve.

One can use `tension` quite intuitively: for a value of 1, the path remains unchanged, for higher values the path gets more and more tense. On the other hand, the operator `controls` gives us absolute control on the curve—but is certainly not intuitive: maybe Leonardo da Vinci was smart enough to be able to guess the control point coordinates of Joconda's smile, but the rest of us would rather be unable to do it.

So it happened that the author often needed "loose" Bézier curves, and was unable to obtain them: unfortunately, `tension` doesn't work with values less[3] than .75. With the following code, one can get arbitrarily loose Bézier curves:

```
def npush(expr p, coef) =
 hide(pair firstpt, firstcpt, secondcpt, secondpt; firstcpt =
postcontrol 0 of p; secondcpt = precontrol 1 of p; firstpt = point 0 of p; secondpt
= point 1 of p; pair intersectpt, newfirstcpt, newsecondcpt;
 intersectpt − firstpt = whatever ∗ (firstcpt − firstpt);
 intersectpt − secondpt = whatever ∗ ( secondcpt − secondpt);
 newfirstcpt = coef[firstcpt, intersectpt]; newsecondcpt = coef[ secondcpt, intersectpt]; )
 firstpt . . controls newfirstcpt and newsecondcpt . . secondpt
enddef;
```

[3] Actually it is not mentioned in the METAFONTbook, what the lower bound of the tension parameter is; repeated tries by the author have shown that the value .75 still works, while $.75 - \epsilon$ produces an error message.

The macro *npush* takes two arguments: the path we want to loosen, and a numerical coefficient. For value 0 of this coefficient, the path remain unchanged. What happens when we increase this value? let's consider the intersection point of the two Bézier tangents (the tangent at curve begin and end). We know that control points always lie on these two tangents. For values between 0 and 1 of the coefficient, the control points travel between their original positions and the intersectionpoint. For value 1 both control points are identified with the intersection point. For values higher than 1 they continue their travel outside of the Bézier triangle.

On Appendix A the reader can see the effects of the *npush* macro applied uniformily to all paths of a circle, with values of the coefficient going from -5 to 5.

As the reader has surely already noticed, this macro doesn't work when the tangents are parallel (because there is no Bézier triangle in that case). A second macro, with a slightly different approach covers all possible cases:

def *mpush*(**expr** p, *lcoef*, *rcoef*) $=$
hide(**pair** *firstpt*, *firstcpt*, *secondcpt*, *secondpt*; *firstcpt* $=$
postcontrol 0 **of** p; *secondcpt* $=$ **precontrol** 1 **of** p; *firstpt* $=$ **point** 0 **of** p; *secondpt*
$=$ **point** 1 **of** p; **pair** *newfirstcpt*, *newsecondcpt*;
newfirstcpt $-$ *firstpt* $=$ *lcoef* $*$ (*firstcpt* $-$ *firstpt*);
newsecondcpt $-$ *secondpt* $=$ *rcoef* $*$ (*secondcpt* $-$ *secondpt*);)
firstpt .. controls *newfirstcpt* and *newsecondcpt* .. *secondpt*
enddef;

This macro has three arguments: the path which we will modify, and two numerical coefficients, corresponding to the transformation at curve begin and at curve end. This time we multiply the distance of the first control point from curve begin by the first coefficient, and the one of the second control point from curve end by the second coefficient. Hence, in this case, value 1 for both coefficients will leave the path unchanged. For values higher than 1 the path will "swell", while for values tending to 0 it will become more and more tense.

These two macros may not be as reliable as primitive METAFONT operators, but they produce easily predictable results and are suitable for fine-tuning of character parts.

3 Getting Text and Numeric Data out of a Font

In his extremely interesting paper on communication between TEX and METAFONT ([**?**]), Alan Hoenig states that ... *METAFONT's file handling abilities are greatly crippled when compared to TEX. Other than font pixel files, font metric files, and log files, METAFONT cannot write files*... To remedy to that situation, we present `GFtoTXT`, a small utility reading text (and any other information) out of GF files produced by METAFONT. As a matter of fact, METAFONT has a *special* command, just like TEX, but until now, no DVI driver was able to use it (as a possible use, one could very well imagine PostScript color instructions embedded in the GF files, taken over by the PK files and translated into real PostScript commands by the DVI driver).

`GFtoTXT` is written in Flex, a UNIX-originated lexical analyzer, under GNU copyleft. Flex allows the generation of highly reliable C code out of simple pattern matching, using regular expressions and states. The Flex code of `GFtoTXT` is very short; the reader can find it in Appendix B. To obtain an executable, (a) run this code through Flex (with the `-8` option, if your Flex version is older than 2.4.6) — Flex will produce a C file called `lex.yy.c` (or `LEXYY.C` on Messy-DOS systems); (b) compile it using your favourite C compiler.[4]

How does it work? There is one convention which must be followed: every string which we want to extract from the METAFONT run, must start with the character `#`, followed by the name of the file on which we are going to write and a colon `:`. The `#` precaution is necessary, because METAFONT itself sends several strings to the GF file by using internal *special* commands. Thse will be ignored by

[4]The Flex code as well as executables can be found on `ftp.ens.fr`, in `pub/tex/yannis/gftotxt`.

GFtoTXT. Hence, to obtain the string "Hello world!" in our output file (let's call it `output.txt`), we will include the command

special("#output.txt:Hello world!");

in the METAFONT code of our file (let's call it `input.mf`). `GFtoTXT` reads from the standard input flow and writes to the files specified in the argument of the *special* command, so we only need to redirect the standard input flow; here is the necessary command line:

```
GFtoTXT < input.2602.gf
```

`GFtoTXT` allows three additional conventions in METAFONT strings: (a) `\n` will produce a carriage return in the output file, and (b) `\x` followed by a two-digit (lowercase) hexadecimal number between `00` and `ff` will produce the corresponding 8-bit character in the output file, (c) `\X` followed by a four-digit (lowercase) hexadecimal number between `0000` and `ffff` will produce the corresponding 16-bit character in the output file. Convention (a) is useful for "formatting" the output file, for example

special("#output.txt:(CHAR C A\n (HEIGHT R "&decimal h&")\n (WIDTH R "& decimal w & ")\n");

will produce

```
(CHAR C A
   (HEIGHT R 99.99976)
   (WIDTH R 129.99683)
```

Convention (b) is useful for inserting 8-bit characters (or characters in the range `"00–"1f`) into the output file. Finally, convention (c) can be useful to those of us who use UNICODE encoding: wouldn't it be nice to have METAFONT say to us ♡Σ' ἀγαπῶ♡? the necessary code would be

special("#output.txt:\X2665\X03a3\X0027\X0020\X1f00\X03b3\X03b1\X03c0\X1ff6\X2665");

4 Bottom Line

The different METAFONT techniques presented in this paper are certainly not programmed in the most elegant way: the author needed them for specific purposes, and stopped testing and refining, whenever the specific problem was solved. The goal of this paper is *not* to provide the reader with powerful new tools, but rather to stimulate him/her in creating his/her own ones, and go beyond the `plain` and `cm` base macros. In all three examples, the basic idea was very simple: iterate calculations until a sufficiently precise approximation is obtained, modify a path by manipulating the control points in the background, read the GF file by something else than GFtoPK or GFtoDVI. By writing down and sharing such ideas we can make out of METAFONT an even friendlier and productive font design tool. To discuss METAFONT relative issues, but also font design in general (and why PostScript and TrueType are less efficient than METAfonts), join the METAFONT e-mail discussion list! The address of the list is

`metafont@ens.fr`

on the Internet. Subscribe by sending the (usual) subscription message

`SUBSCRIBE METAFONT` *`Your name and institution`*

to `listserv@ens.fr`.

References

[1] Hoenig, A. *When TeX and METAFONT talk: Typesetting on curved paths and other special effects*, pp. 549–553, TUGboat **12** (4), 1991

A Transforming a circle through *npush*