MFLua: Instrumentation of MF with Lua

Abstract
We present MFLua, a METAFONT version which is capable of code instrumentation and has an embedded Lua interpreter that allows glyphs curves extraction and post-processing. We also show and discuss an example of a METAFONT source processed by MFLua to output an OpenType font.

1 Introduction

MFLua is a version of METAFONT, Knuth's program (Knuth, 1986b) designed to draw fonts. MFLua has an embedded Lua interpreter, as well as the capability of the Pascal-WEB code instrumentation to output information about bitmaps and curves used in glyphs drawing. The latest capability is known as *code tracing*. MFLua's main goal is to ease the production of vector fonts which source code is a METAFONT code. MFLua doesn't extend the METAFONT language in any way (i.e., it's not possible to embed Lua code in a METAFONT source file), so that a METAFONT source file is fully compatible with MFLua and vice versa. MFLua won't be extended like LuaTEX extends pdfLaTEX. The code instrumentation is a facility to gather and manage information collected in the log file when METAFONT `tracing` instructions are enabled. MFLua automatically saves data into Lua tables using external Lua scripts. Therefore a programmer can manage these tables according to his needs, i.e. extracting a glyph vector outline(s). Rephrasing the previous statements, MFLua is a (bitmap) *tracing* program that knows curves in advance instead of determining them from the bitmap. Please notice that this is only possible when the data have been gathered.

The paper has the following structure: after explaining what *code instrumentation* is (section 2), it shows the components used to trace a glyph (section 3) and finally two different approaches to manage curves (section 4).

As a final remark, we consider MFLua as being in a state between a proof of concept and alpha and it's not (yet) too user-friendly. Its code is hosted at `https://github.com/luigiScarso/mflua`.

2 Code instrumentation

METAFONT is written in Pascal-WEB (a programming language by Donald Knuth to have a real literate programming tool. As the name suggests, it's a subset of Pascal) and is automatically translated into C by `tangle` and `web2c`. Instrumentation is the capability to add *trace statements* (a.k.a. *sensors*) in strategic points of the code to register current state information and pass it to the Lua interpreter. A typical sensor has the `mflua` prefix. We can see some sensors in the following chunk of code, the main body of METAFONT (slightly refolded to fit the printing area).

```
@p begin @!{|start_here|}
mflua_begin_program;
{in case we quit during initialization}
history:=fatal_error_stop;
t_open_out; {open the terminal for output}
if ready_already=314159 then goto start_of_MF;
@<Check the ``constant'' values...@>@;
if bad>0 then
  begin wterm_ln('Ouch---my internal constants
     have been clobbered!',
    '---case ',bad:1);
@.Ouch...clobbered@>
  goto final_end;
  end;
{set global variables to their starting values}
initialize;
@!init if not get_strings_started then
  goto final_end;
init_tab; {initialize the tables}
init_prim; {call |primitive| for each primitive}
init_str_ptr:=str_ptr; init_pool_ptr:=pool_ptr;@/
max_str_ptr:=str_ptr; max_pool_ptr:=pool_ptr;
fix_date_and_time;
tini@/
ready_already:=314159;
mfluaPRE_start_of_MF;
start_of_MF: @<Initialize the output routines@>;
@<Get the first line of input and prepare
  to start@>;
history:=spotless; {ready to go!}
mflua_initialize;
if start_sym>0 then
  {insert the `\&{everyjob}' symbol}
  begin cur_sym:=start_sym; back_input;
  end;
mfluaPRE_main_control;
main_control; {come to life}
mfluaPOST_main_control;
final_cleanup; {prepare for death}
mfluaPOST_final_cleanup;
end_of_MF: close_files_and_terminate;
final_end: ready_already:=0;
end.
```

We're going to examine the role of the mflua_begin_program sensor. The Pascal-into-C translator, web2c, is smart enough to distinguish a symbol already present in the Pascal source from an external symbol (i.e., a symbol defined in another file). In the latter case, the programmer has to register that symbol into the file texmf.defines if the symbol is related to an argumented procedure: the translator will manage properly the arguments translation. The translated code will contain the C form of the sensor symbol, which will be resolved at compile-time — i.e., we need an object file that contains that symbol. Each sensor is stored into mflua.h and mflua.c. The first one lists the symbol:

```
#include "lua51/lua.h"
#include "lua51/lualib.h"
#include "lua51/lauxlib.h"
#include <kpathsea/c-proto.h>
#include <web2c/config.h>

extern lua_State* Luas[];
extern int mfluabeginprogram();
```

while the second one contains the corresponding function source code:

```
int mfluabeginprogram()
{
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  Luas[0] = L;
   /* execute Lua external "begin_program.lua" */
  const char* file = "begin_program.lua";
  int res = luaL_loadfile(L, file);
  if ( res==0 ) {
      res = lua_pcall(L, 0, 0, 0);
    }
  priv_lua_reporterrors(L, res);
  return 0;
}
```

The above function initializes the Lua interpreter, stores its state in the array Luas[] (it would be possible to have more than one interpreter but this feature is currently neglected) and then executes the external script begin_program.lua calling lua_pcall(L, 0, 0, 0). This call protects the interpreter from errors. Every time we run mf (the METAFONT program), it loads and executes the Lua script begin_program.lua, customizable by programmers.

We surely need to pay attention to some issues. The first one is that literate programming style allows to collect every changes we make in a source file into a *change* file (mf.ch in our case), which is then merged into a Pascal program by tangle. This means that inserting a sensor can interfere with the change file. In this case we also have to insert the sensor into the change file as we do, for instance, with mfluaPRE_make_ellipse(major_axis, minor_axis,theta,tx,ty,0). Of course the right solution is directly inserting the sensors in the change file. Unfortunately it's usually faster discovering where to insert a sensor in the source file and then managing conflicts in the change file: source files have a context — the source itself — that change file don't. The second issue is the need to export some METAFONT variables and constants to the Lua interpreter. An easy way to accomplish this task is inspecting the translated C code to realize how those variables and constants are managed. For example, to make Lua read the charcode variable, which contains the index of the current glyph, we need to know that it's stored into the internal array at index 18 (the index is from the METAFONT WEB source) so that we can write a wrapper function like the following one:

```
#define roundunscaled(i) (((i>>15)+1)>>1)
EXTERN scaled internal[maxinternal + 1]  ;
static int
 priv_mfweb_LUAGLOBALGET_char_code(lua_State *L)
{
  integer char_code=18;
 integer p=roundunscaled(internal[char_code])%256;
 lua_pushnumber(L,p);
  return 1;
}
```

and then make it available to the Lua interpreter as the LUAGLOBALGET_char_code variable:

```
int mfluainitialize()
{
  /* execute Lua external "mfluaini.lua" */
  lua_State *L = Luas[0];
  /* register lua functions */
:
lua_pushcfunction(L,
       priv_mfweb_LUAGLOBALGET_char_code);
lua_setglobal(L,"LUAGLOBALGET_char_code");
:

/* execute Lua external "mfluaini.lua" */
  const char* file = "mfluaini.lua";
  int res = luaL_loadfile(L, file);
  if ( res==0 ) {
      res = lua_pcall(L, 0, 0, 0);
  }
  priv_lua_reporterrors(L, res);
  return 0;
}
```

Users can read and set this variable though the set value won't be passed to METAFONT in order to interfere as little as possible with its state. That's why we prefer to inspect the translated C code, which quality depends on the translation performed at compile-time. A clean solution should only depend on the WEB source. For historical reasons, translating code from Pascal into C outputs two files (mf0.c and mf1.c). Finding a symbol implies searching in two files, which hardens the process.

There are currently 24 sensors, 33 global variables and 15 scripts, though it's possible to increase these quantities if we discover that tracing a specific function inside METAFONT is better than reimplementing it in Lua. While it's easy to implement the algorithm to draw a curve in Lua, it's slightly harder to implement the algorithm to fill a region. Whatever, the main goal is to keep the number of sensors as low as possible. Notice that MFLua currently reads scripts from the current folder and doesn't use the standard TEX folders.

The counterpart of `mflua_begin_program` is `mflua_end_program`, which calls the `end_program.lua` script. It contains all the functions used to transform the components of a glyph, the subject of the next section.

3 The components of a glyph

METAFONT mainly manages Bézier cubic curves (see fig. 1).[1] This curve is completely described by four *controls points*: $\mathbf{p}$ (called the *starting point*), $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{q}$ (known as the *ending point*). The METAFONT command to draw such a curve is

```
draw p .. controls c₁ and c₂ .. q;
```

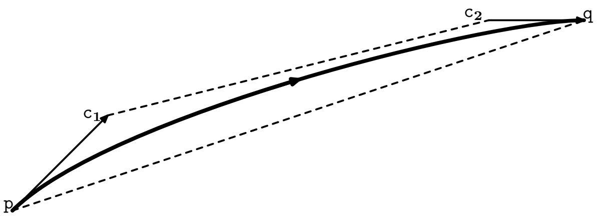


Figure 1. A cubic Bézier curve and its convex hull.

This curve lies on the XY plane and its *parametric form* is quite simple:

$$\mathbf{B}(t) = (1-t)^3\mathbf{p} + 3(1-t)^2t\mathbf{c_1} + 3(1-t)t^2\mathbf{c_3} + t^3\mathbf{q} \qquad \forall t \in [0,1] \quad (1)$$

The corrensponding algebraic expression, the *closed form*, is more complex but it's useful to quickly test whether a point belongs to the curve or not.

Equation (1) has first derivatives $\mathbf{B}'(0) = 3(\mathbf{c_1} - \mathbf{p})$ and $\mathbf{B}'(1) = 3(\mathbf{q} - \mathbf{c_2})$ respectively when $t = 0$ and $t = 1$. We can easily calculate them when we know $\mathbf{p}$, $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{q}$. An important property is that a Bézier cubic curve is completely contained in the polygon $\mathbf{p}\,\mathbf{c_1}\,\mathbf{c_2}\,\mathbf{q}\,\mathbf{p}$ (the convex hull) and this immediately leads to the conclusion that the intersection of two curves is empty if and only if the intersection of their convex hulls is empty. Another important property is the existence of the *De Casteljau's algorithm*, very easy to implement: given the four control points of a curve and a time t_1, it returns the point $(x_1, y_1) = \mathbf{B}(t_1)$ on the curve and the two series of control points, one for the curve $\mathbf{B_l}(t) = \mathbf{B}(t), t \in [0, t_1]$ (the *left side*) and one for the curve $\mathbf{B_r}(t) = \mathbf{B}(t), t \in [t_1, 1]$ (the *right side*). It recursively reduces the curve $\mathbf{B}(t)$, $t \in [0,1]$ tracing to the tracing of the left side $\mathbf{B_l}(t) = B(t)$, $t \in [0, 1/2]$ and the right side $\mathbf{B_r}(t) = B(t)$, $t \in [1/2, 1]$ (the recursion ends when the distance between two points (x_j, y_j) and (x_{j+1}, y_{j+1}) is less than a pixel).

The De Casteljau's algorithm is useful because it estimates how long a curve is counting the number of pixels covered by the curve. It also finds intersections of two curves $\mathbf{B}(t)$ and $\mathbf{C}(t)$ reducing this problem to the problem of calculating the intersection of four curves: left and right side of $\mathbf{B}(t)$ and left and right side of $\mathbf{C}(t)$ for $t = 1/2$. The algorithm keeps working when one curve degenerates into a segment (i.e., if $\mathbf{p} = \mathbf{c_1}$ and $\mathbf{c_2} = \mathbf{q}$) or when it degenerates into a point ($\mathbf{p} = \mathbf{c_1} = \mathbf{c_2} = \mathbf{q}$). Therefore it can be used to find an intersection between a line and a curve and to test if a point belongs to a curve. A set of curves $\{\mathbf{B}_1, \mathbf{B}_2 \ldots \mathbf{B}_n\}$ where $\mathbf{q}_{j-1} = \mathbf{p}_j$ and $\mathbf{q}_n = \mathbf{p}_0$ is a simple cycle if the only intersection is $(x, y) = \mathbf{q}_n = \mathbf{p}_0$. Simple cycles are the building blocks of a glyph: a simple cycle can be filled or unfilled and, according to METAFONT's point of view, a glyph is a set of cycles filled and/or unfilled at the right moment.

A normal METAFONT designer doesn't care about these details because METAFONT has a high level language to describe curves, points, lines, intersections, filled and unfilled cycles and, most important, pens. The listed entities produce a combination of two different basic draws: regions (un)filled by a *contour* and regions (un)filled by the stroke of a pen, i.e., the *envelope of a pen*. Both are simple cycles, but their origin is very different.

Let's consider the code of the glyph 0 from the file `xmssdc10.mf`:

```
cmchar "The numeral 0";
beginchar("0",9u#,fig_height#,0);
italcorr fig_height#*slant-.5u#;
adjust_fit(0,0);
penpos1(vair,90);
penpos3(vair,-90);
penpos2(curve,180);
penpos4(curve,0);
if not monospace:
 interim superness:=sqrt(more_super*hein_super);
fi
x2r=hround max(.7u,1.45u-.5curve);
x4r=w-x2r; x1=x3=.5w;
y1r=h+o; y3r=-o;
y2=y4=.5h-vair_corr;
y2l:=y4l:=.52h;
penstroke pulled_arc.e(1,2) & pulled_arc.e(2,3)
 & pulled_arc.e(3,4)
 & pulled_arc.e(4,1) & cycle;  % bowl
penlabels(1,2,3,4);
endchar;
```

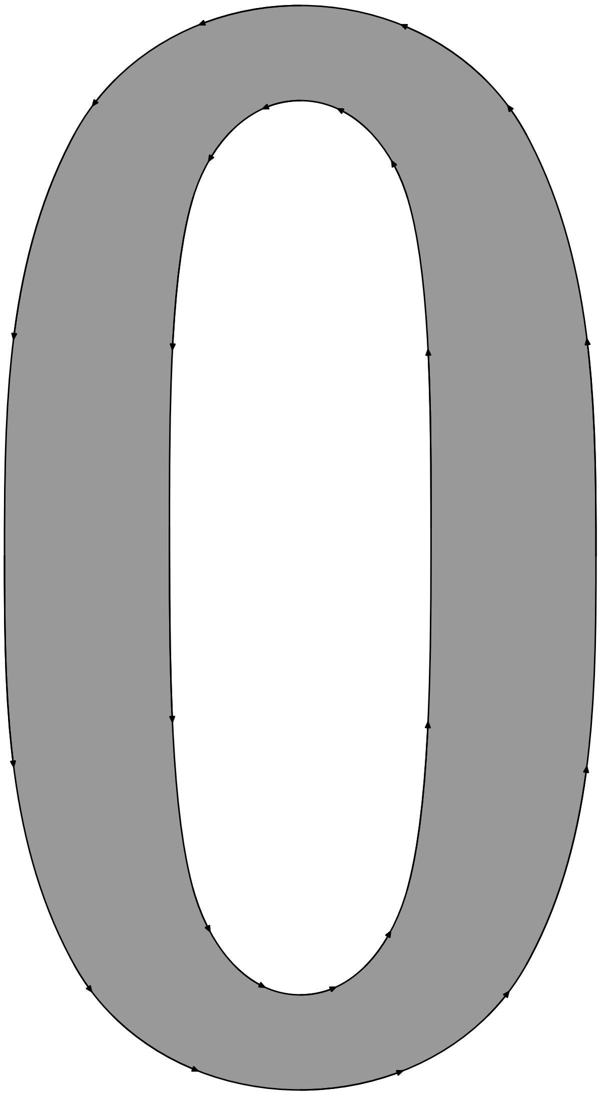


Figure 2. The glyph of the numeral 0 in `xmssdc10.mf` font.

Fig. 2 shows a glyph only made by two contours which are the result of `penpos` and `penstroke` macros. Of course we could obtain the same result drawing 24 curve sections (12 for the outer contour, 12 for the inner one) but it should be clear that the METAFONT description is much more straight or, at least, "typographic".

Things completely change when we consider the numeral 2:

```
cmchar "The numeral 2";
beginchar("2",9u#,fig_height#,0);
italcorr fig_height#*slant-.5u#;
adjust_fit(0,0);
numeric arm_thickness, hair_vair;
hair_vair=.25[vair,hair];
arm_thickness=
Vround(if hefty:slab+2stem_corr
      else:.4[stem,cap_stem] fi);
pickup crisp.nib;
pos7(arm_thickness,-90); pos8(hair,0);
bot y7r=0; lft x7=hround .9u; rt x8r=hround(w-.9u);
y8=good.y(y7l+beak/2)+eps;
arm(7,8,a,.3beak_darkness,beak_jut);%arm and beak
pickup fine.nib; pos2(slab,90);
pos3(.4[curve,cap_curve],0);
top y2r=h+o; x2=.5(w-.5u);
rt x3r=hround(w-.9u); y3+.5vair=.75h;
if serifs:
 numeric bulb_diam;
 bulb_diam=hround(flare+2/3(cap_stem-stem));
 pos0(bulb_diam,180); pos1(cap_hair,180);
 lft x1r=hround .9u; y1-.5bulb_diam=2/3h;
 (x,y2l)=whatever[z1l,z2r];
 x2l:=x; bulb(2,1,0);  % bulb and arc
else: x2l:=x2l-.25u; pos1(flare,angle(-9u,h));
 lft x1r=hround .75u; bot y1l=vround .7h;
 y1r:=good.y y1r+eps; x1l:=good.x x1l;
 filldraw stroke term.e(2,1,left,.9,4);
fi  % terminal and arc
pos4(.25[hair_vair,cap_stem],0);
pos5(hair_vair,0);
pos6(hair_vair,0);
y5=arm_thickness; y4=.3[y5,y3];
top y6=min(y5,slab,top y7l);
lft x6l=crisp.lft x7;
z4l=whatever[z6l,(x3l,bot .58h)];
z5l=whatever[z6l,z4l];
erase fill z4l--
 z6l--lft z6l--
 (lft x6l,y4l)--cycle;%erase excess at left
filldraw stroke z2e{right}..tension
  atleast .9 and atleast 1
 ..z3e{down}..{z5e-z4e}z4e--z5e--z6e;%stroke
penlabels(0,1,2,3,4,5,6,7,8);
endchar;
```

As we can see in fig. 3, there are both a contour and envelopes of more than a pen; there are intersections between the contour the envelopes and the pens, and some curves are outside the glyph (some of these curves are used to delete unwanted black pixels). There are also some unexpected straight lines and small curves. The number of curves looks quite large, which is not what we desire as we want to obtain the outline depicted in fig. 4.

Unfortunately, things are even different and it's necessary to describe how METAFONT calculates pen envelopes to go on. This is explained in the book "METAFONT: The Program" (Knuth, 1986a) at the "Polygonal pens" part, chapter 469, that we briefly quote with a slightly modified notation:

> Given a convex polygon with vertices $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{n-1}, \mathbf{w}_n = \mathbf{w}_0$ a in *counter-clockwise* order …(and a curve $\mathbf{B}(t)$) the envelope is obtained if we offset $\mathbf{B}(t)$ by $\mathbf{w}_k$ when the curve is travelling in a direction $\mathbf{B}'(t)$ ly-

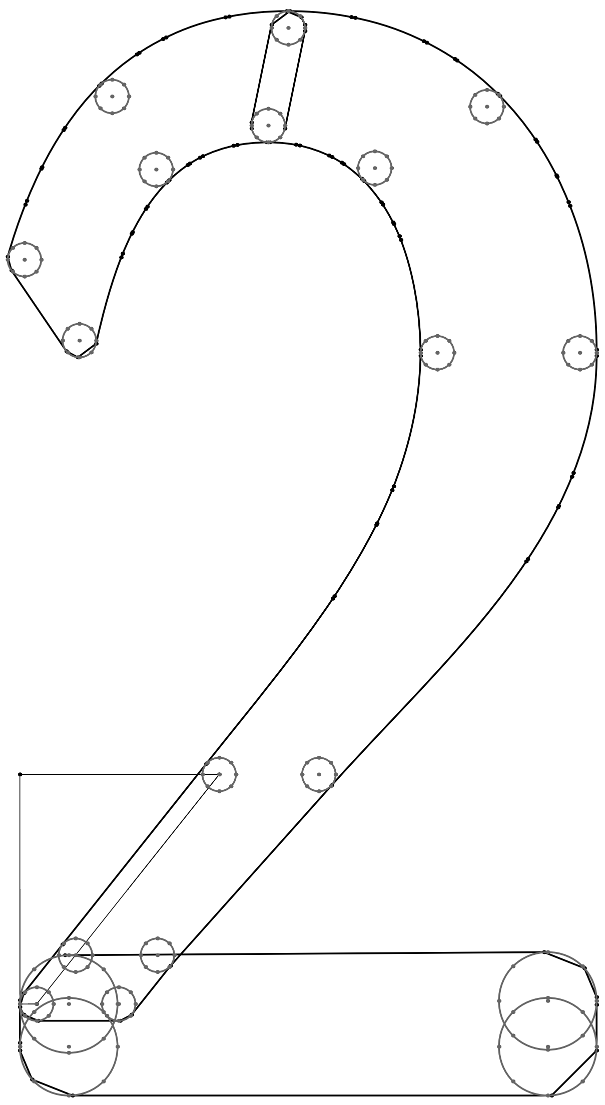


Figure 3. The glyph of the numeral 2 in xmssdc10 font. We can see envelopes and pens (thick curves) and a contour (thin curve).

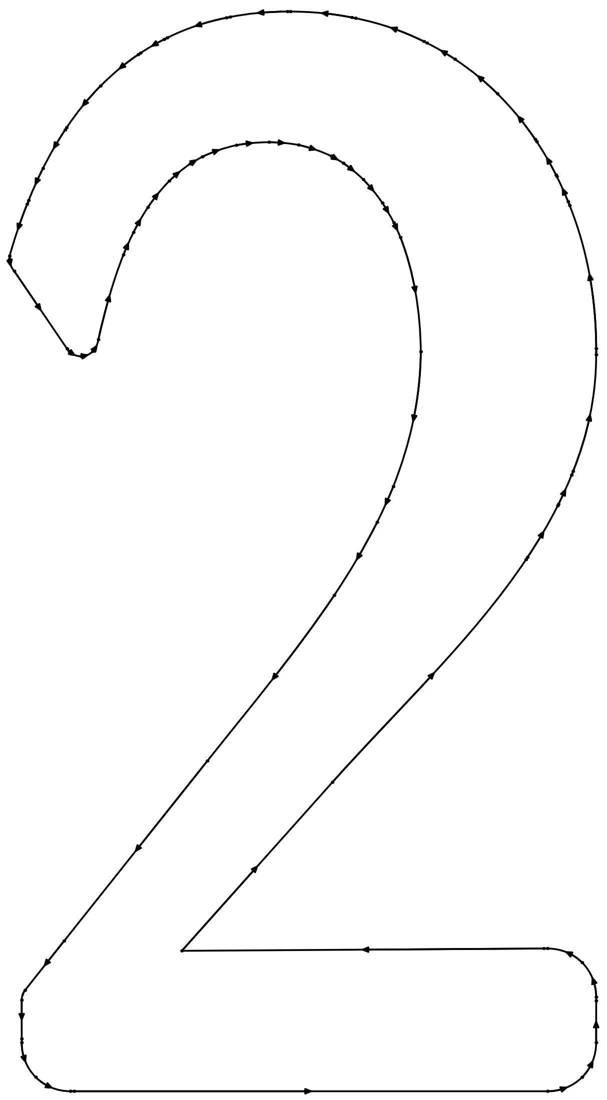


Figure 4. An outline of the numeral 2 in xmssdc10.mf font.

ing between the directions $\mathbf{w}_k - \mathbf{w}_{k-1}$ and $\mathbf{w}_{k+1} - \mathbf{w}_k$. At times t when the curve direction $\mathbf{B}'(t)$ increases past $\mathbf{w}_{k+1} - \mathbf{w}_k$, we temporarily stop plotting the offset curve and we insert a straight line from $\mathbf{B}(t) + \mathbf{w}_k$ to $\mathbf{B}(t) + \mathbf{w}_{k+1}$; notice that this straight line is tangent to the to the offset curve. Similarly, when the curve direction decreases past $\mathbf{w}_k - \mathbf{w}_{k-1}$, we stop plotting and insert a straight line from $\mathbf{B}(t) + \mathbf{w}_k$ to $\mathbf{B}(t) + \mathbf{w}_{k+1}$; the latter line is actually a "retrograde" step which will not be part of the final envelope under the METAFONT's assumptions. The result of this construction is a continuous path that consist of alternating curves and straight line segments.

This explains why the number of the curves is large and why there are small curves, but says nothing about those circular curves that we can see in fig. 4: METAFONT indeed converts an elliptical pen into a polygonal one and then applies the algorithm. The conversion is accurate enough to guarantee that the envelope is correctly filled with the right pixels. This is a key point to understand: *METAFONT's main task is to produce the best bitmap of a glyph, not the best outline.*

The role of the sensors is to gather as much information as possible about pixels, contours, the polygonal version of the pens, envelopes and their straight lines and then store these information (basically the edge structure of the pixels and Bézier curves with an eventual offset) into appropriate Lua tables. As METAFONT halts, the Lua interpreter calls `end_program.lua` and let the programmer manage these tables: sometimes, as we have seen in the numeral 0 case, the post-process can be quite simple, sometimes not. MFLua doesn't automatically output a glyph outline because it's the programmer who has to implement the best strategy according to his experience.

4 Two different strategies for post-processing the curves

The Concrete Roman 10 pt

The first use of MFLua has been the post-processing of Concrete Roman 10 pt to obtain an OpenType version of it. This font is described in the file `ccr10.mf`. As we previously said, sensors collect the data into Lua tables and `end_program.lua` post-processes them at the end of the execution (we could even choose to execute the no-more post-process during the execution). The script `end_program.lua` defines the global array `chartable[index]` that contains the data for the glyph with char code `index`: we have the edge structure that allows the program to calculate the pixels of the glyph as well as the three arrays `valid_curves_c`, `valid_curves_e` and `valid_curves_p` that gather the data of contours, envelopes and the polygonal version of the pens. Each array contains the array of the control points `{p,c1,c2,q}` stored as a string `"(<x>,<y>)"`, where `<x>` and `<y>` are the coordinates of the point. With fig. 3 as a reference, we can see that when we draw a glyph with a pen it usually has overlapping strokes. Along with the curves of the pen(s), these overlaps create curves inside or outside the glyph that must be deleted. Having the pixels of the glyph, we can use the parametric form (1) to check if a point (x, y) (or better, a neighborhood with center (x, y)) is inside or outside. If all the points of the curve are inside or outside, we can delete them. The drawback is that while time t goes linearly in $\mathbf{B}(t)$, the points $(x(t), y(t))$ follow a cubic (i.e., not linear) law in case the curve is not a straight line. Hence, they are not equally spaced — this means that we can jump over some critical points. Using the same time interval steps for each curve means that short curves are evaluated in times where the points can differ less than a pixel — a useless evaluation. Of course not all the curves are inside the glyph: there are curves on the border and curves partially lying on the border and partially inside (or outside). In the latter case the result of evaluation is an array of time intervals where the curve crosses the border.

Once we have deleted the curves that are completely inside (or outside) the glyph, the next step is to merge all the curves and split them using the previously seen time interval (this is done by a Lua implementation of the De Casteljau's algorithm). Now we have a set of curves that are on the border or "near" it (i.e., partially on the border). We can delete those curves having only one intersection (a *pending curve*), supposing that each curve of the final outline has exactly 2 intersections at times $t_0 = 0$ and $t_1 = 1$.

To calculate all the intersections we use the following trick: if we have n curves, we produce a METAFONT file that contains the code that calculates the intersection between p_i and p_j for $1 \leq j \leq n$ and $j < i \leq n$ (given that $p_j \cap p_i = p_i \cap p_j$) and then we parse the log file with Lua. For example if

```
p1={"(57.401,351.877)", "(57.401,351.877)",
    "(57.901,349.877)", "(57.901,349.877)"}
```

and

```
p2={"(56.834,356.5)",   "(56.834,354.905)",
    "(57.031,353.356)", "(57.401,351.877)"}
```

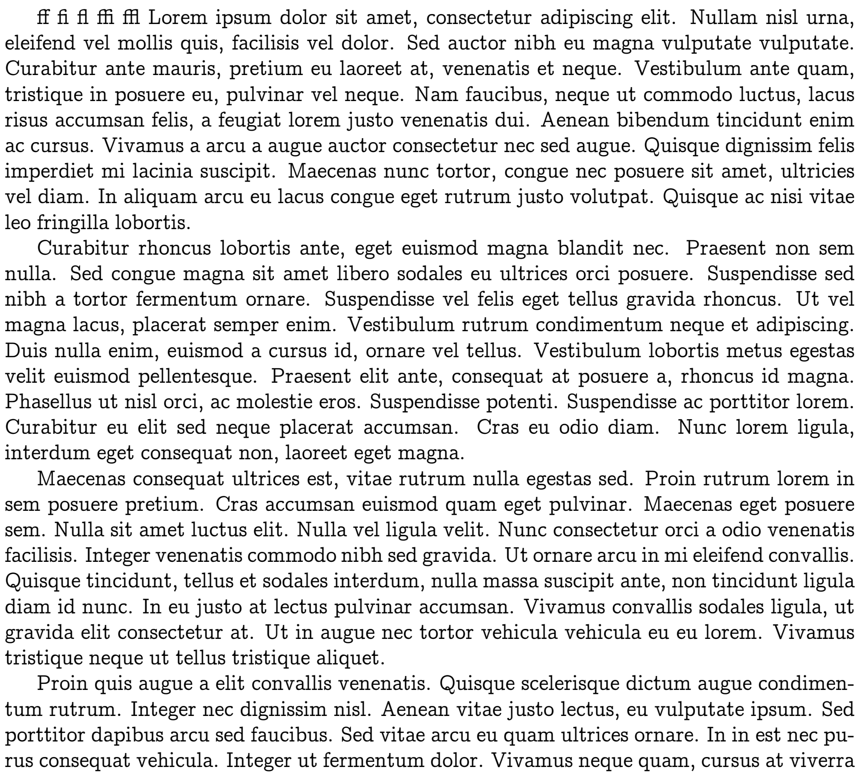
then we have

```
batchmode;
message "BEGIN i=2,j=1";
path p[];
p1:=(57.401,351.877) ..
  controls (57.401,351.877) and (57.901,349.877) ..
    (57.901,349.877);
p2:=(56.834,356.5) ..
  controls (56.834,354.905) and (57.031,353.356) ..
    (57.401,351.877);
numeric t,u;
(t,u) = p1 intersectiontimes p2;
show t,u;
message "" ;
```

and the log

```
BEGIN i=2,j=1
>> 0
>> 0.99998
```

If the result is $(-1, -1)$ the intersection is empty. There are two problems with this approach: the first one shows when a curve crosses the border and time intervals can generate two curves, one completely outside and one completely inside — hence deleting an intersection. To avoid this issue we must adjust the intervals moving the extremes a bit. We have the second problem when there can be curves with three or more intersections — i.e., we can have *loops*. Opening a loop can be a difficult task: e.g., if the curve p_a intersects $\{p_b, p_c, p_d\}$ at the same time t_a and p_b intersects $\{p_a, p_c, p_d\}$ at t_b then $I_a = \{p_a\} \cup \{p_b, p_c, p_d\}$ is equal to $I_b = \{p_b\} \cup \{p_a, p_c, p_d\}$ and we can delete p_a and p_b because p_c and p_d stay connected. But with more than three intersections things become more complex.


ff fi fl ffi ffl Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam nisl urna, eleifend vel mollis quis, facilisis vel dolor. Sed auctor nibh eu magna vulputate vulputate. Curabitur ante mauris, pretium eu laoreet at, venenatis et neque. Vestibulum ante quam, tristique in posuere eu, pulvinar vel neque. Nam faucibus, neque ut commodo luctus, lacus risus accumsan felis, a feugiat lorem justo venenatis dui. Aenean bibendum tincidunt enim ac cursus. Vivamus a arcu a augue auctor consectetur nec sed augue. Quisque dignissim felis imperdiet mi lacinia suscipit. Maecenas nunc tortor, congue nec posuere sit amet, ultricies vel diam. In aliquam arcu eu lacus congue eget rutrum justo volutpat. Quisque ac nisi vitae leo fringilla lobortis.

Curabitur rhoncus lobortis ante, eget euismod magna blandit nec. Praesent non sem nulla. Sed congue magna sit amet libero sodales eu ultrices orci posuere. Suspendisse sed nibh a tortor fermentum ornare. Suspendisse vel felis eget tellus gravida rhoncus. Ut vel magna lacus, placerat semper enim. Vestibulum rutrum condimentum neque et adipiscing. Duis nulla enim, euismod a cursus id, ornare vel tellus. Vestibulum lobortis metus egestas velit euismod pellentesque. Praesent elit ante, consequat at posuere a, rhoncus id magna. Phasellus ut nisl orci, ac molestie eros. Suspendisse potenti. Suspendisse ac porttitor lorem. Curabitur eu elit sed neque placerat accumsan. Cras eu odio diam. Nunc lorem ligula, interdum eget consequat non, laoreet eget magna.

Maecenas consequat ultrices est, vitae rutrum nulla egestas sed. Proin rutrum lorem in sem posuere pretium. Cras accumsan euismod quam eget pulvinar. Maecenas eget posuere sem. Nulla sit amet luctus elit. Nulla vel ligula velit. Nunc consectetur orci a odio venenatis facilisis. Integer venenatis commodo nibh sed gravida. Ut ornare arcu in mi eleifend convallis. Quisque tincidunt, tellus et sodales interdum, nulla massa suscipit ante, non tincidunt ligula diam id nunc. In eu justo at lectus pulvinar accumsan. Vivamus convallis sodales ligula, ut gravida elit consectetur at. Ut in augue nec tortor vehicula vehicula eu eu lorem. Vivamus tristique neque ut tellus tristique aliquet.

Proin quis augue a elit convallis venenatis. Quisque scelerisque dictum augue condimentum rutrum. Integer nec dignissim nisl. Aenean vitae justo lectus, eu vulputate ipsum. Sed porttitor dapibus arcu sed faucibus. Sed vitae arcu eu quam ultrices ornare. In in est nec purus consequat vehicula. Integer ut fermentum dolor. Vivamus neque quam, cursus at viverra

Figure 5. The ConcreteOT font produced by MFLua from `ccr10.mf`.

To solve these cases, `end_program.lua` has a series of *filters*. A filter acts on a specific glyph and typically removes unwanted curves and/or adjusts the control points to ensure that a curve joins properly with its predecessors and successor. Of course this means that the programmer inspects each glyph separately, which is reasonable when we are designing the font — less reasonable when we convert it.

We can call this approach *per-font and per-glyph*: `end_program.lua` is a Lua script valid only for a specific font and which has filters for each glyph.

The script `end_program.lua` also has some functions to convert the outlines (with the correct turning number) of each glyph into a SVG font: this font format can be imported into FontForge and, usually after re-editing the glyphs (typically simplifying the curves), it can be saved as an OpenType CFF. In fig. 5 we can see an example of this font.

The Computer Modern Sans Serif Demibold Condensed 10 pt

We now approach a more "geometric" strategy. We don't want to output an OpenType font but to find an `end_program.lua` more *universal and per-glyph* and less *per-font and per-glyph*. Our experience with `ccr10.mf` make us believe that is always possibile to write a METAFONT program that outputs a nice bitmap of a glyph using a very complex set of curves. This is especially true when we use pens and the need to manually correct every error arises. Up to now we only made few outlines of numerals.

There are new functions to trace a curve and to calculate the intersections between two cubics (both based on De Casteljau's bisection algorithm, an application of De Casteljau's algorithm) so the parametric form and the trick to calculate the intersections are not needed anymore. We also keep contours, envelopes and pens apart almost until the end of the process, when we first merge envelopes and pens and then, at last, contours. The most important enhancement is probably the replacement of the polygonal version of a pen with an elliptical one. METAFONT generates a polygonal getting an ideal ellipse with major axis, minor axis and the angle of rotation from the pen specifications and then calls `make_ellipse`. Putting a sensor around helps us store the axis and theta into a Lua table, to be read

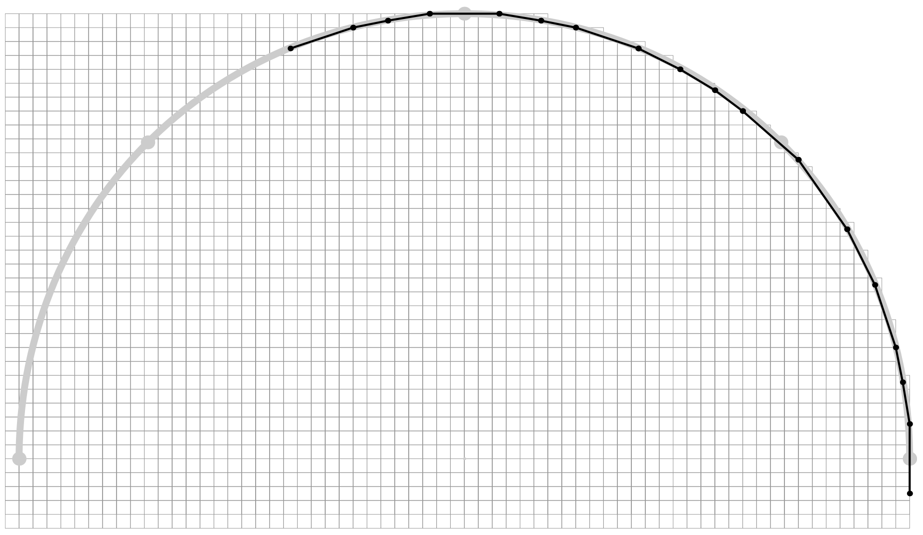


Figure 6. Real polygonal pen (black) vs. calculated elliptical pen (gray). Square boxes are the pixels. It's the bottom right part of fig. 3 .

later from `end_program.lua`. The next step is a trick again: we call MFLua with the following file:

```
batchmode;
fill fullcircle
    xscaled (majoraxis)
    yscaled (minoraxis)
    rotated (theta) shifted (0,0);
shipit;
bye.
```

where `majoraxis`, `minoraxis` and `theta` get the ellipse data. MFLua then saves the outlines of the filled ellipse into another file, from which they can be read by `end_program.lua`. This script then saves each elliptical pen in a table, with `p..c1..c2..q` as a key to be reused later, instead of the polygonal one. We can see the result in fig. 6: the approximation is quite good. This reduces the total number of curves and gives the glyph a more natural look.

5 Conclusion

We believe that MFLua is an interesting tool for font designers because too many fonts (if not all) are currently designed using contours. In this case `end_program.lua` should be simple (less or even no intersections, compared to the METAFONT technique, see the numeral 0 of `xmssdc10.mf`). On the other side, using the pens shows that extracting an outline is a difficult task. It's almost impossible to find an always valid script. The outlines from an envelope usually have a large number of curves, which is not a good feature, and this is a METAFONT property: we can always implement routines to simplify them, though FontForge already does it.

The work will continue on `xmssdc10.mf` to find an `end_program.lua` modular and flexible enough for a wide application.

References

KNUTH, D. E. (1986a). *METAFONT: The Program.* Addison-Wesley, Massachusetts, 1ª edizione.

— (1986b). *The METAFONTbook.* Addison-Wesley, Massachusetts, 1ª edizione. — with final correction made in 1995 —.

MARSH, D. (2005). *Applied Geometry for Computer Graphics and CAD.* Springer, London, 2ª edizione.

Notes

1. I borrow notation from MARSH (2005), where points and functions in the Bézier curves section are represented by bold, upright letters.

Luigi Scarso
luigi dot scarso at gmail dot com