Converting METAFONT sources to outline fonts using METAPOST

Karel Píška
Institute of Physics, Academy of Sciences
182 21 Prague
Czech Republic
piska@fzu.cz
http://www-hep.fzu.cz/~piska/

Abstract

The paper describes a multistep conversion process from METAFONT sources to outline fonts (Adobe Type 1 format). An important step, finding contours, is based on an accurate algorithm fitting the envelope curve of a stroke drawn by a pen along a cubic Bézier curve by the least square method, specially extended (adapted) for a rotated elliptical pen applied, for instance, in the Devanagari font design. After converting the EPS files produced by METAPOST to the corresponding outline representation, the FontForge font editor is used for removing overlap, simplification, autohinting, generating outline fonts, and necessary manual modifications. The conversion results, the faithful Indic Type 1 fonts (significantly more precise and closer to optimal than earlier attempts made by autotracing bitmaps) will be released.
Keywords: font conversion, bitmap fonts, METAFONT, METAPOST, outline fonts, PostScript, Type 1 fonts, approximation, Bézier curves.

Introduction

In 2001 I experimented with approximate conversion METAFONT Indic fonts to the Type 1 format by autotracing bitmaps with the TEXtrace program [11]. I was not satisfied with the results and decided to apply another, analytic approach, to achieve results more precise and also better optimized.

Conversion process

Our procedure consists of studying the font definitions in METAFONT and preparing encoding files. Then the glyph strokes produced by METAPOST are converted to outlines, and the font is assembled, optimized, and autohinted. Finally, it is generated as a Type 1 binary file with FontForge. After verification of visual proofsheet pages some steps are often repeated to correct or improve the final results.

Analysis of METAFONT sources We analyze the METAFONT source texts [7] of a font to select an appropriate strategy of conversion, to find the crucial parameters, like the font size, the italic angle, and the definitions of pens and strokes. Some parameters may be also hidden inside macros. Sometimes, a method for efficient conversion is not apparent. Therefore it is also important to know about the presence and number of METAFONT commands not available in METAPOST [5], for example, using operations with bitmap picture variables.

Creating encoding files Encoding files and encoding vectors define a mapping between the glyph names and their number codes. METAFONT definitions usually do not contain unique glyph names in an explicit form but only as comments. The glyph names are taken from these comments to produce an unambiguous list of PostScript names, i.e. we must find duplicated names and change them to be different. Our preliminary solution inherits the METAFONT comments closely to make glyph identification easier.

Running METAPOST Invoking METAPOST processes the METAFONT sources and produces EPS files. METAPOST together with a macro package `mfplain` ([5], p. 79) allows processing the original or modified (to eliminate METAFONT-specific commands) font sources written in METAFONT and to generate for each glyph a single file in the Encapsulated PostScript format, consisting only of PostScript commands like curves, strokes, affine transformations representing pens, etc., but no bitmap images, in contrast to METAFONT's usual output. Some metric data, e.g. the glyph widths and italic angles, may be lost; we shall restore them later.

Figure 1: Result of METAPOST.

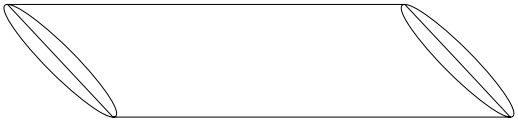

Figure 2: Primary conversion to outlines.

We also need to define a magnification factor, because we have to transform the glyph images to a 1000-unit glyph coordinate system (we use this usual space) with the units in PostScript big points (*bp*) and the font *designsize* in *pt* units. The transformation factor from *pt* to *bp* is 1.00375. Thus in general the magnification factor will be $1000 * 1.00375/designsize$. For *designsize* = 10 pt, this is $1000 * 1.00375/10 = 100.375$, for 8 pt it is 125.46875, for 17.28 pt 58.087384, etc.

So, a typical command to call METAPOST is:

```
mpost '&mfplain \mode=localfont;' \
      mag=100.375'; input' dvng10.mf
```

These files may contain various stroked paths (see figures 1, 9). It is necessary to find contour curves for single strokes and then also common envelope curves for overlapping strokes.

The following lines from the PostScript produced by METAPOST correspond to fig. 1:

```
 0 79.06227 dtransform truncate idtransform
 setlinewidth pop [] 0 setdash
 1 setlinecap 1 setlinejoin 10 setmiterlimit
gsave newpath 119.50958 284.54501 moveto
398.36119 284.54501 lineto
[-0.98387 0.98387 -0.17888 -0.17888 0 0] concat
stroke grestore
```

The `lineto` operator describes the line segment, the `concat` operator applies the affine transformation represented by the preceding normalized matrix (in brackets) denoting the rotated elliptical pen, and `79.06227 ... setlinewidth` is the scale factor defining the stroke width.

Converting METAPOST products to outlines The results of METAPOST (strokes) are converted to "primary" outlines. Fitting curves with the least square method is a typical approach to calculate a curve approximation. This method is nothing new and may well have been used in conversion programs developed by Richard Kinch (MetaFog, [6]), Basil Malyshev [9], George Williams (FontForge, [13]) and others. We only apply a few additional conditions. We try to be more precise, but our attempts are still more fragile and unstable than the programs listed above.

All the calculations are in the non-integer value space. We check each segment for accuracy and subdivide it if a chosen limit is exceeded; insert all horizontal and vertical extrema nodes; keep all horizontal/vertical straight lines and control vectors to be exactly horizontal/vertical. The inner part of a contour curve of drawing a rotated elliptical pen even along a simple Bézier path without any intersections may have self-intersections. Therefore we try to find self-intersection points as precisely as possible, if it is possible at all. Unfortunately, sometimes this iteration does not converge. A simplest conversion to outlines is shown in figure 2.

For a given time of the path segment using the affine transformation matrix and its inverse matrix (for a usual pen they are always regular) we can calculate the displacement corresponding to the point lying on the right parallel outline curve (the left one is located symmetrically). Knowing the coordinates of points on the outline curves and also on the pen boundary we can fit them by a cubic Bézier approximation. But a problem is that we do not know whether the points are on the envelope curve, because parts of the outline curves may create loops of arbitrary size being inside a closed area. It depends on complex correlations between the path and the pen.

We also recognize quarter-circles, usually represented in METAFONT by two segments because METAFONT tends to divide curves into octants. To avoid further simplification problems, we do not preserve the 45 degree middle nodes and change the quarter-circles to the accurate single-segment PostScript representation with relative lengths of control vectors $4/3(\sqrt{2}-1) \simeq 0.552285$, cf. R. Kinch [6, p. 236] and Luc Devroye [2]. For an example of our circle approximation see figure 3.

To summarize, in the primary approximation straight lines and circles are represented by the minimal number of segments (because other nodes are unnecessary), and, on the other hand, other outline curves have redundant node points (to preserve a maximal starting accuracy). These intermediate results of the primary conversion to outline are demonstrated in figures 2 and 10.

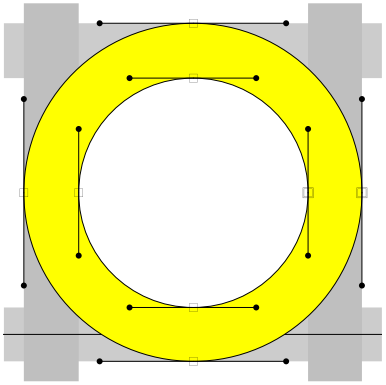

Figure 3: Representation of circles.

Creating a font with FontForge FontForge is a powerful open source font editor. Among its wide range of useful abilities is a "background layer", which may contain bitmap images and line drawings. So, we generate with METAFONT high resolution bitmaps for the font under study, either 2400 dpi (`supre` mode), or 7254 dpi. The 7254 dpi "device" corresponds to the relation 1 pixel in the PK bitmap $\sim$ 1 unit in the PostScript glyph space for a 10 pt design size:

```
% (72.27*1000.375/10dpi)=7254.1
mode_param (pixels_per_inch,4000+3254.1);
mode_param (blacker, 0);
mode_param (fillin, 0);
mode_param (o_correction, 1);
```

The resulting GF (or PK) files can be imported to the background as a set of gray pixels to compare with glyph images. (Sometimes, METAFONT with such a high resolution may fail, if the author did not design the font for an arbitrary resolution.)

Font composition We also run mftrace [10] with an appropriate encoding to make a PFB font file. From this file we build a frame for the created font, copy the glyph widths and glyph names, and move the outlines to the background layer (visible as green lines). During a subsequent processing of the font with FontForge we use its internal Spline Font Database format (SFD).

The high resolution bitmap is always huge, so we import it only before a comparison. But the outline contours of the font produced by mftrace are not large and we can store them in the working SFD files permanently. To the foreground layer we import the outlines from the EPS files calculated in the previous step from the original EPS files generated by METAPOST.

The high resolution pixel image gives a close visual bitmap representation of the original METAFONT source. Of course, information about contour curves, intersection points, corners, etc., virtually calculated by METAFONT has been lost. The font outlines autotraced by mftrace from similar bitmaps, despite the inevitable artifacts (bumps, holes, unrecognized corners, ...) give reasonably correct information about the glyphs. Our aim is to obtain another outline representation: more accurate and closer to optimal, minimizing the number of defects.

Having the font in SFD format built from the mftrace output, our next step with FontForge is **removing overlap and optimization** (simplification). We continue processing in the non-integer space to keep accuracy, in particular, not changing the slopes of the neighboring control vectors so as to preserve a smooth transition between segments.

Rounding to integer, hinting and Type 1 font generation FontForge allows for generating PostScript fonts with non-integer point coordinates and, PostScript RIP devices (usually) render these fonts properly. But we have three significant reasons to round coordinates to integers and to generate the Type 1 fonts in integer representation:

- Non-integer values in the PostScript charstring occupy 3 "items". Therefore the integer representation saves storage and the PFB files are smaller.
- The final Type 1 fonts do not need such accuracy after removing overlap and simplification.
- For hinting it would be inconvenient and impractical to use any discrete grid other than integers.

For example, the non-integer Type 1 command occupies 19 items:

```
18153 100 div 212 100 div
14437 100 div -407 100 div
7208 100 div -243 100 div
rrcurveto
```

and after rounding only 7 items:

```
182 2 144 -4 72 -2 rrcurveto
```

It is desirable to minimize the number of items because PostScript interpreters have internal memory limits per glyph. Exceeding these limits causes a **limitcheck** error and rendering fails.

The coordinates of the segments are rounded to integers by a more complex algorithm than a trivial rounding of all the values. First, we round the node points. Then we transform the control vectors according the changes of the nodes, and try to find the control points in the integer grid near the transformed control vectors. Even this sophisticated rounding to integer is not without problems. Sometimes, if the change in x or y in the segment is very

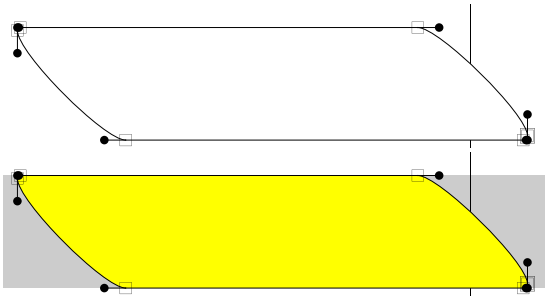

Figure 4: Final font in an outline form and a hinted proofsheet (clip).

small (e.g. about 1 unit) or a segment is too short (in both directions) no good selection may exist and a manual adjustment is then necessary, probably with the loss of closeness, accuracy or symmetry of the approximation.

No special additional program for hinting has been developed or applied. The automatic autohinting tool in FontForge is used, and any unsatisfactory results should be corrected manually.

Finally, we generate the Type 1 binary font with FontForge, rounded to integer coordinates and (auto)hinted as described. See figure 4.

Results

To make font auditing and verification quicker and more efficient, we have developed tools for generation of visual proofsheets in PDF. These support a fast overview of all glyph images, display of outline curves with node and control points and vectors, and hinting zones. They can also produce warnings about undesirable situations such as missing nodes at extrema, presence of inflection inside a segment, a non-smooth transition between segments, etc.

Our aim is to fulfill the *Type 1 conventions* [1]. Therefore we include the extrema nodes (they may be omitted if they are really redundant), exclude other unnecessary node points, and preserve smooth connections between the adjacent segments. We also keep the straight lines, corners and arcs after conversion, and avoid any false bumps, holes or steps absent in the original METAFONT sources.

In this paper, some selected figures have the node points (squares), the control points (bullets) and the control vectors enlarged for readability. In a real working process they are colored and small as in other proofsheets, when we zoom in on interesting details only if we need to check them.

The principal and auxiliary algorithms are still under development and adaptation for new fonts.

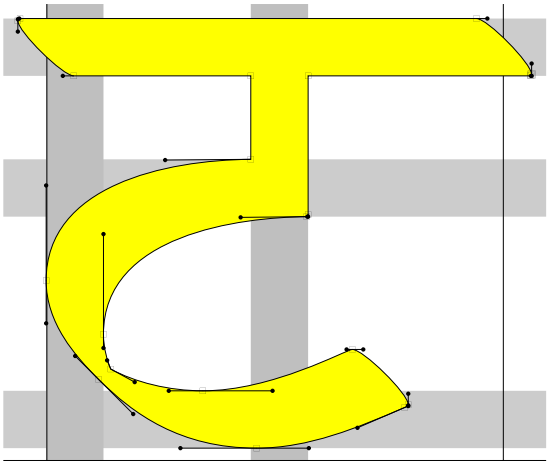

Figure 5: `dvng10`: tta of Frans Velthuis.

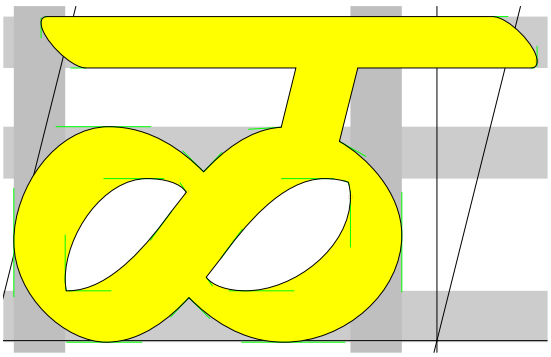

Figure 6: `dvngbi10`: lla of Frans Velthuis.

The programs are written in `awk` or `gawk` [3]. For Type 1 font handling the `t1utils` [8] are used.

Several pictures illustrate the intermediate and final results in the conversion of METAFONT fonts to the Type 1 format: figures 2, 4, 10, 11, 15, and 16.

Indic fonts A basic goal of the work is more precise outline versions of the free METAFONT Indic fonts available from CTAN: Devanagari, Sanskrit, Gurmukhi, Punjabi, Bangla, Sinhala, Malayalam, Telugu, Kannada, Tamil, and Tibetan. At the time of writing, not all the above fonts have been converted. Also, the Oriya fonts are impractical because they widely use METAFONT bitmap picture commands.

Figures 5, 6, 12, 13 (all Devanagari), and 14 (for Malayalam) show the results to date.

Chinese fonts We have also tried to convert two small single fonts with Chinese ideographs created in METAFONT: the Hóng-Zì font (128 glyphs) designed by Javier Rodríguez Laguna [12] (version 0.5 of 2005-03-23), shown in fig. 7; and `china10`, a font from the `china2e` package [4] containing Chinese

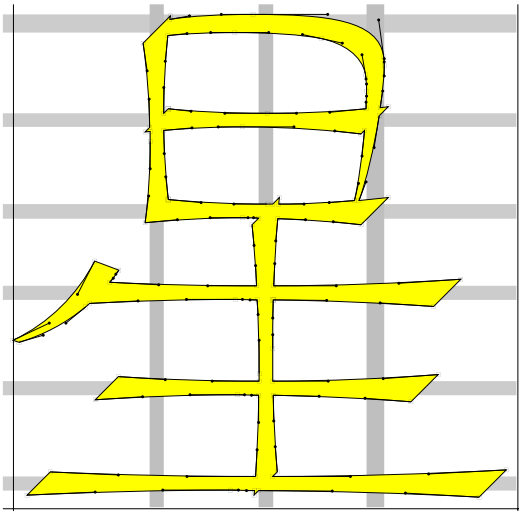

Figure 7: Hóng-Zì: xing1 of Javier Laguna.

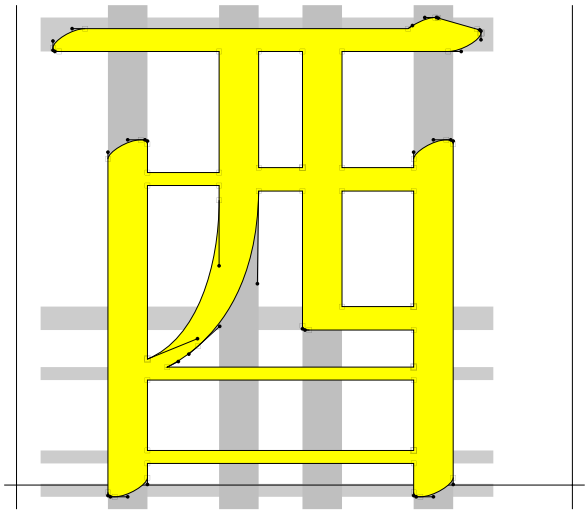

Figure 8: china10: yeu of Udo Heyl.

calendar symbols produced by Udo Heyl (1997), in fig. 8.

Conclusion

In this article we describe a font conversion process and briefly discuss some selected problems. Creating *precise* fonts is always difficult, time-consuming and never-ending work, regardless of the approach chosen. We plan to again verify all the glyphs to improve hinting and polish the outlines to remove tiny artifacts. It will also be useful to make the glyph names of the Indic glyphs common for all languages; this is not trivial because the fonts contain many various ligatures, special signs or variants not covered in the Unicode standards.

Acknowledgements

I would like to thank all the authors of the free conversion programs, the authors of the public METAFONT fonts for Indic languages, other sources and program packages used in this work.

References

[1] Adobe Systems Inc. *Adobe Type 1 Font Format.* Addison-Wesley, 1990.

[2] Luc Devroye. "Formatting Font Formats", *TUGboat* **24**(3), pp. 588–596, 2003.

[3] Free Software Foundation. GNU awk, http://www.gnu.org/software/gawk.

[4] Udo Heyl. china2e, CTAN:macros/latex/contrib/china2e, 1997.

[5] John D. Hobby. *A user's manual for METAPOST.* AT&T Bell Laboratories, Computing Science Technical Report 162, 1994.

[6] Richard J. Kinch. "MetaFog: Converting METAFONT Shapes to Contours", *TUGboat* **16**(3), pp. 233–243, 1995.

[7] Donald E. Knuth. *The METAFONTbook.* Addison-Wesley, 1986. Volume C of *Computers and Typesetting.*

[8] Eddie Kohler. t1utils: Type 1 font utilities, http://freshmeat.net/projects/t1utils.

[9] Basil K. Malyshev, "Problems of the conversion of METAFONT fonts to PostScript Type 1", *TUGboat* **16**(1), pp. 60–68, 1995.

[10] Han-Wen Nienhuys. mftrace, http://www.cs.uu.nl/~hanwen/mftrace.

[11] Karel Píška. "A conversion of public Indic fonts from METAFONT into Type 1 format with TEXtrace." *TUGboat* **23**(1), pp. 70–73, 2002.

[12] Javier Rodríguez Laguna. Hong-Zi: a Chinese METAFONT, http://hongzi.sourceforge.net, 2005.

[13] George Williams. FontForge: an outline font editor, http://fontforge.sourceforge.net.

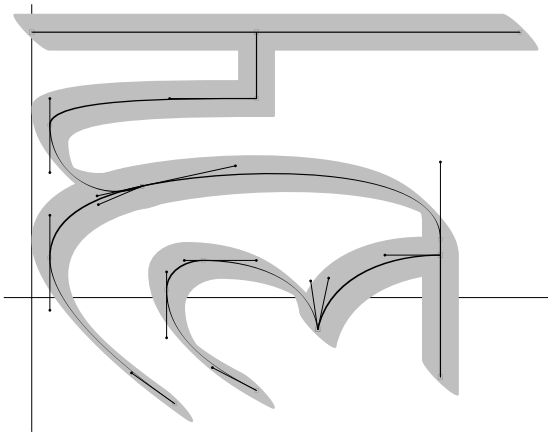

Figure 9: dvng10 l_h: METAPOST output.

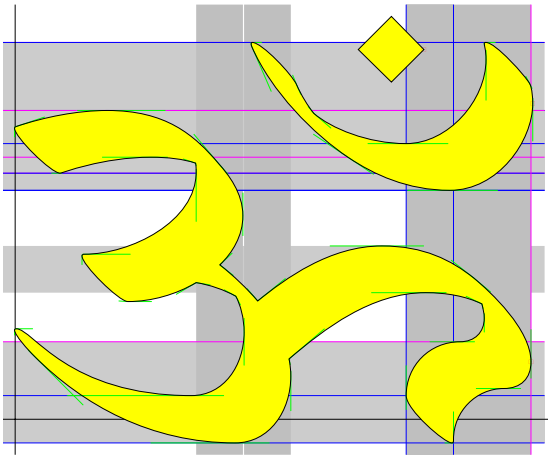

Figure 12: dvng10: om of Frans Velthuis.

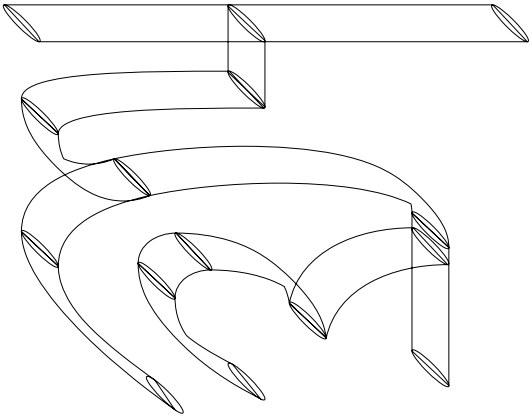

Figure 10: dvng10 l_h: primary outlines.

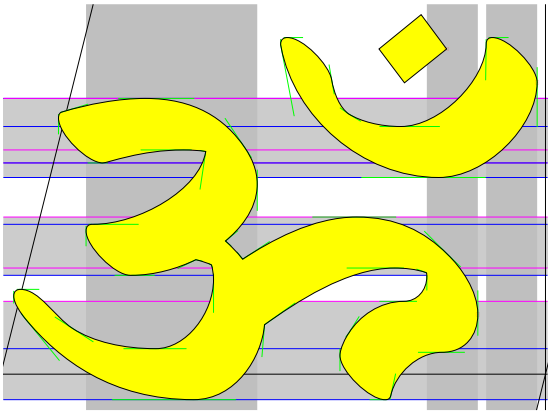

Figure 13: dvngbi10: om of Frans Velthuis.

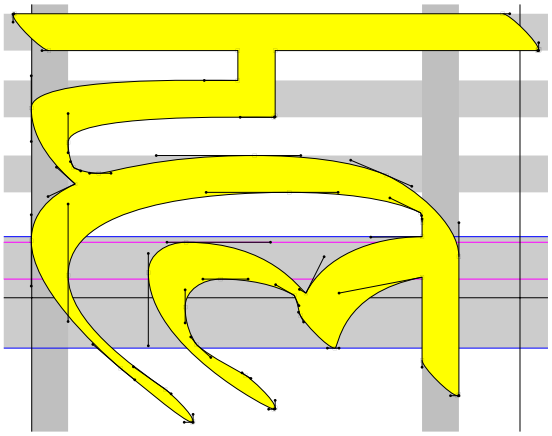

Figure 11: dvng10 l_h: Type 1 font proofsheet.

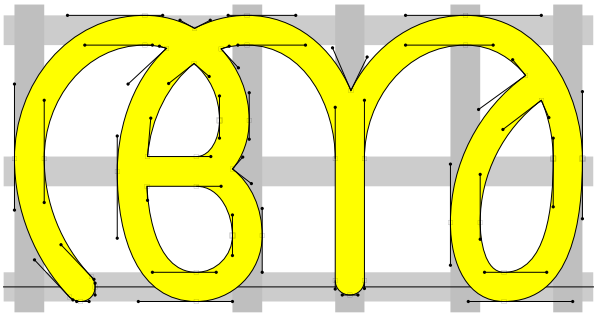

Figure 14: mm10: a of Jeroen Hellingman.

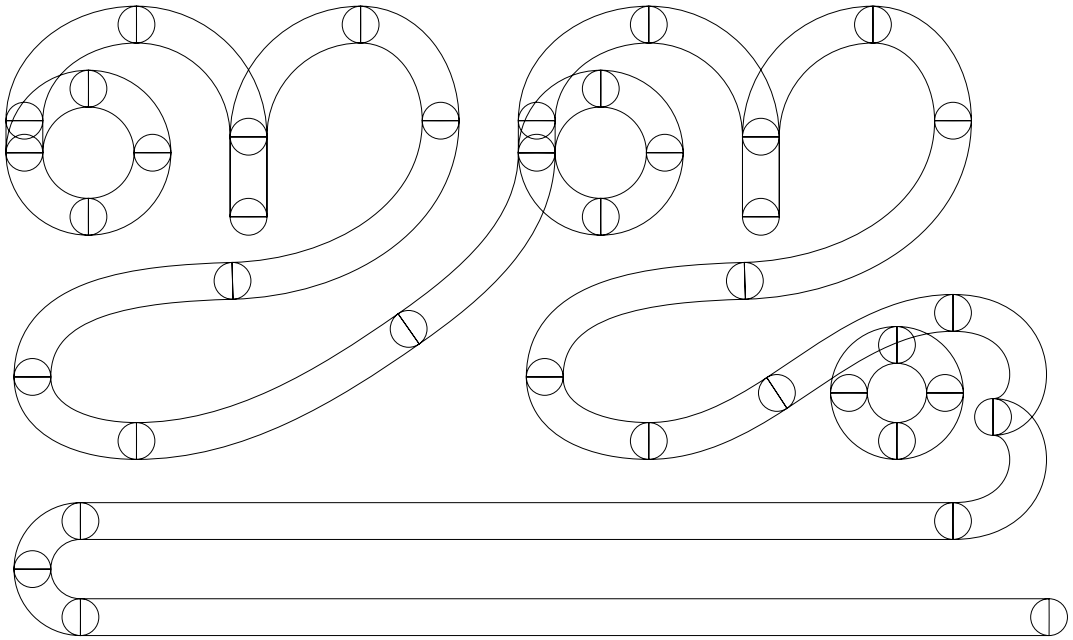

Figure 15: mm10 j_juu: METAPOST output converted to primary outlines.

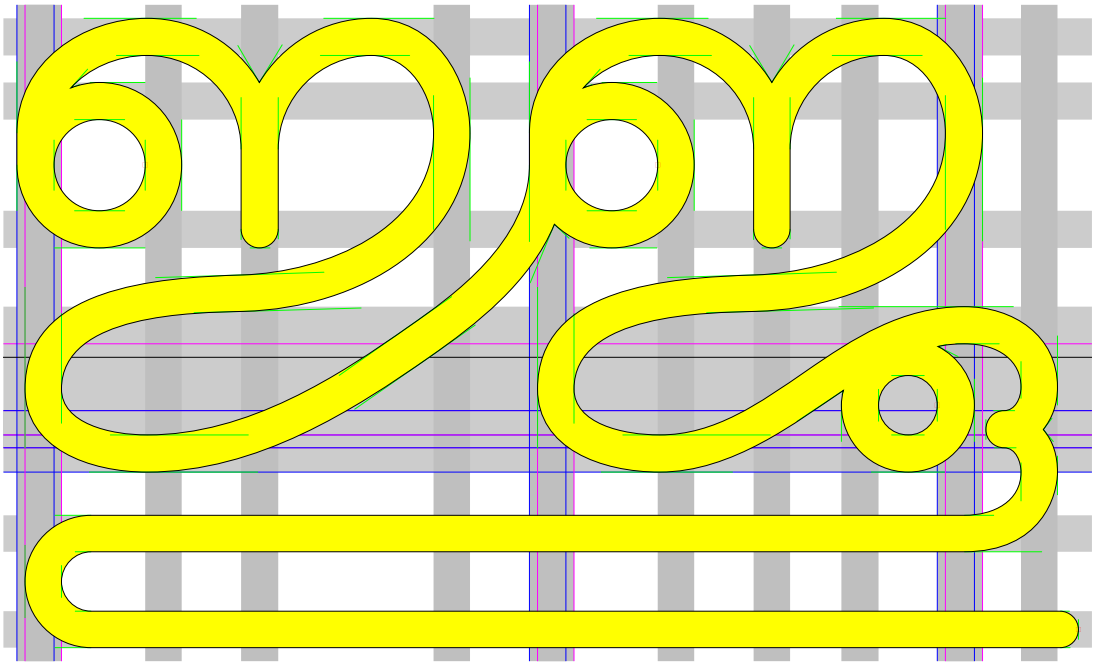

Figure 16: mm10 j_juu: Type 1 font proofsheet with hints.