Environment for Translating METAFONT to POSTSCRIPT

Shimon Yanai and Daniel M. Berry

Abstract

This paper describes a program, mf2ps, that translates a METAFONT font definition into a definition for the same font in the POSTSCRIPT language. mf2ps is constructed out of the part of the METAFONT program that extracts the envelopes of the letters; these envelopes are converted into POSTSCRIPT outlines.

1 Introduction

This paper describes a program, mf2ps, that takes from a METAFONT [10, 11] program for a font all the necessary information in order to create an equivalent POSTSCRIPT [1] font definition. The program makes use of the front end of the METAFONT program to extract the envelopes of the letters to produce the POSTSCRIPT outlines. What makes this process natural is that both METAFONT and POSTSCRIPT make liberal use of Bézier curves to describe non-circular curves.

By producing this translator, it is hoped to be able to produce from METAFONT fonts POSTSCRIPT outline fonts which are more compact than the bitmapped fonts produced by the METAFONT program. Certainly the outline fonts are more easily scaled to other magnifications and possibly even other design sizes than are bitmaps. Moreover, doing so makes fonts heretofore available only on TEX [9] and other DVI-based formatters, available on ditroff [8] and other formatters which have evolved, or have been designed, for use with POSTSCRIPT printers. This paper, which is typeset by ditroff, uses a POSTSCRIPT version of the logo font in order to print the word "METAFONT" in the same appearance as in TEX-generated documents. Moreover, these new POSTSCRIPT outline fonts can be used in TEX also! One needs only the TEXpS [3] software.

The organization of this paper is as follows. Section 2 presents the background of this work. Section 3 explains the rationale behind building the translator and describes a previous attempt at writing the translator and an approach to avoid. The software engineering aspect of the translator is described also in Section 3. The details of the implementation are exposed in Section 4. Section 5 describes the operation of the program. Section 6 evaluates the results. Finally Section 7 describes improvements to the translator that are left for future work.

2 Background

Typesetter formatting systems such as TEX and ditroff use fonts as raw material. The formatters accept mixed text and commands as input and produce output, which, if sent to the laser printers or typesetters, yields formatted text printed on pages. The laser printers and typesetters use fonts, i.e., sets of printable patterns, one per character, in various representations in order to cause the desired characters to appear on the printed form. For some printers, bitmaps are used, with 1's representing inked dots and 0's representing non-inked dots. Other printers accept commands that cause drawing of the characters, the printer providing the inked dots according to the drawing commands. One such popular command language is POSTSCRIPT, and its usual use is to specify the outline of the character with the interpreting printer filling in the outline with ink. One popular method of describing fonts is with the METAFONT language, in which declarative definitions of how to paint the characters are given in terms of pen path and pen shape. Another popular method is the same POSTSCRIPT that many printers accept. The prime difference is that the METAFONT program translates the font definitions into bitmaps prior to sending the font to the printer while a POSTSCRIPT printer translates the outlines into bitmaps at the time of printing. Interestingly, both the METAFONT language and the POSTSCRIPT language use Bézier curves for describing the curves followed by the pen or the outlines. As usually configured these days, TEX uses bitmapped fonts in the Computer Modern family generated by METAFONT, and ditroff uses POSTSCRIPT outline fonts supplied by Adobe.

The subsequent subsections delve deeper into these issues in order to be able to state the goal of this paper in the next section.

2.1 Fonts, design sizes, and magnifications. As mentioned, fonts are the raw material of typesetting. A font is a set of printable patterns, one for each character, that causes printing of that character in a particular recognizable style on the page. As mentioned, these patterns can be represented by bitmaps or drawing instructions.

Characters come in various sizes. There are two independent notions of sizing for fonts, point size or design size and magnification. The *design size* is the size at which the character is designed to be used and is, in well-designed text, the size in which the character appears in final, printed copy. Design size is usually expressed in units of points, which are each approximately 1/72 of an inch. Most normal text in books, newspapers, and magazines is printed in 10 point type. Headlines are larger, perhaps as large as 30 points. The *magnification* of a font is the inverse of the ratio between the design size of the character and the size of the character as it emerges on the printer, the assumption being that the final copy is a photo reduction of the printed copy. Thus, if photo reduction halves linear dimensions, one prints with magnification 2. If everything is done right, then after reduction, the letter appears at its design size.

A 10 point design sized font printed at magnification 2 is similar to but not quite the same as a 20 point version of the same font. For example, the serifs on a large point size are smaller than they would be if strict linear magnification were used. Other proportions, e.g., of x-height to cap-height and of width to height, are also different. While many purists, Knuth included, insist on using a different pattern for each design size, many people accept magnification as yielding acceptable fonts at other point sizes. If the unit of magnification is not too big the results are acceptable even to many purists.

2.2 Problems with bitmapped fonts. A bitmap for a character is a rectangular array of bits covering the so-called bounding box or frame that exactly contains a letter. Figure 1 shows a low resolution bit map for the letter "N" in a sans serif font. The inked squares or pixels are denoted by "1" bits and the uninked pixels are denoted by "0" bits.

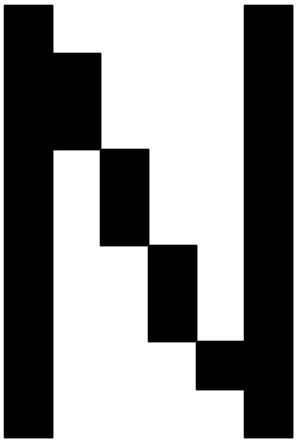

Figure 1

The low resolution example of Figure 1 illustrates a major problem with bitmapped fonts. Curved lines and straight lines that are neither vertical nor horizontal cannot be represented exactly by a rectangular pattern of pixels. One is forced to approximate them with rectangular steps. At high resolution, e.g. above 1000 or so, the human eye cannot see the steps, but at low resolution the steps are quite apparent. Visible steps are called "jaggies" after the jagged edges.

Bitmaps for a font must be built for each design size, magnification, and resolution. If the resolution is fixed, as is the case on most printers, a bitmap must be built for each design size and magnification. An attempt to use a given bitmap at a larger design size or magnification by just enlarging the area of each dot yields a bad case of jaggies.

2.3 METAFONT and its environment. METAFONT, a language for the specification of fonts or typefaces, has been used to provide fonts for the TEX family of typesetting systems. A METAFONT user writes a program for each letter or symbol of an alphabet. These programs are different from the usual computer programs, because they are essentially declarative rather than imperative, using an algebraic language to describe the center stroke or edges of the characters. The description of a letter in METAFONT is a set of equations describing the strokes. When combined with parameters describing the pen shape and size, one gets a full description of a letter. Sizes and shapes of pen nibs can be varied in METAFONT and the characters can be built up in such a way that the outlines of each stroke are precisely controlled. Herein lies the advantage of METAFONT; a font is easily specified and variations are obtained by varying parameters.

Currently, the program that converts a set of METAFONT font descriptions into a bitmapped font translates the description of a letter combined with a point size and a magnification into a bitmap. This bitmap can be sent to the printer to get a letter on the page. Herein lies a disadvantage of METAFONT; a bit map must be kept for each point size and magnification, and this can require a lot of space.

2.4 The POSTSCRIPT language. The POSTSCRIPT language is an interpretive programming language with graphics capabilities. POSTSCRIPT's extensive page description capabilities are embedded into a general-purpose programming language framework. The language includes a conventional set of data types such as numbers, arrays, and strings, control primitives such as conditionals, loops and procedures, and some unusual features such as dictionaries. In most POSTSCRIPT fonts, each letter is described by an imperative program tracing the outline of the letter. This tracing may include curves given as Bézier curves, straight lines, arcs, etc. A POSTSCRIPT printer interprets this outline program to draw and fill in the letters on the page. Some consider the imperative nature of POSTSCRIPT to be a disadvantage in comparison to META–

FONT's declarative nature. The main advantage of POSTSCRIPT relative to METAFONT is that one needs to keep only the outline. If, as in the usual case, the outline is specified in terms of a fixed path through Euclidean two-space, this outline may be scaled arbitrarily to yield any magnification. The scaling is done by the POSTSCRIPT interpreter at the printer. Thus the different magnifications do not require any additional storage space. Actually, the outlines are kept as if they were for the Adobe-standard 1000 dots per emm, which at a design size of 10 points amounts to 7200 dpi. Because a typical phototypesetter has a maximum resolution of about 2500 dpi, the outlines are said to be arbitrarily scaleable. If the outlines are kept, as are many METAFONT definitions, as paths through points calculated by the outline program, then it is possible to, say, make serifs grow more slowly than linearly. It would then be possible to have one POSTSCRIPT font scaleable to all design sizes. Generally, outline fonts are not written this way, so that strictly speaking they are scaleable only to all magnifications.

In addition, the POSTSCRIPT language has a way to work with bitmapped fonts. While the POSTSCRIPT printer can scale them before printing, the end result is that each of the fixed number of dots in the bitmap is made larger or smaller. Since the human will see larger dots as jagged lines, such fonts are not really considered scaleable.

2.5 Bézier curves. Both METAFONT and POSTSCRIPT use Bézier cubics to specify curves. For the Bézier form, four points are used, the start point, the end point, and two control points, as shown in the top half of Figure 2. The tangent vectors of the endpoints are determined from the line segments P_1P_2 and P_3P_4. The mathematical introduction of the Bézier form when given four points P_1, P_2, P_3, and P_4 is

$$z(t) = (1-t)^3 P_1 + 3t(t-1)^2 P_2 + 3t^2(1-t) P_3 + t^3 P_4,$$

for $0 \le t \le 1$.

Two characteristics of the Bézier form tend to make it widely used in graphics. First, by choosing the control points one can easily mold the curve to a desired shape. Second, the four control points taken in another order define a convex polygon, P_1 P_2 P_4 P_3 P_1 in this case, the *convex hull,* which bounds the Bézier curve. The convex hull is useful in clipping a curve against a window.

When a METAFONT user specifies a path, METAFONT creates a list of knots and control points for the associated cubic spline curves. If the user has not specified the control points explicitly, METAFONT itself finds some for the splines of a curve, while POSTSCRIPT requires all the four points to be explicitly given.

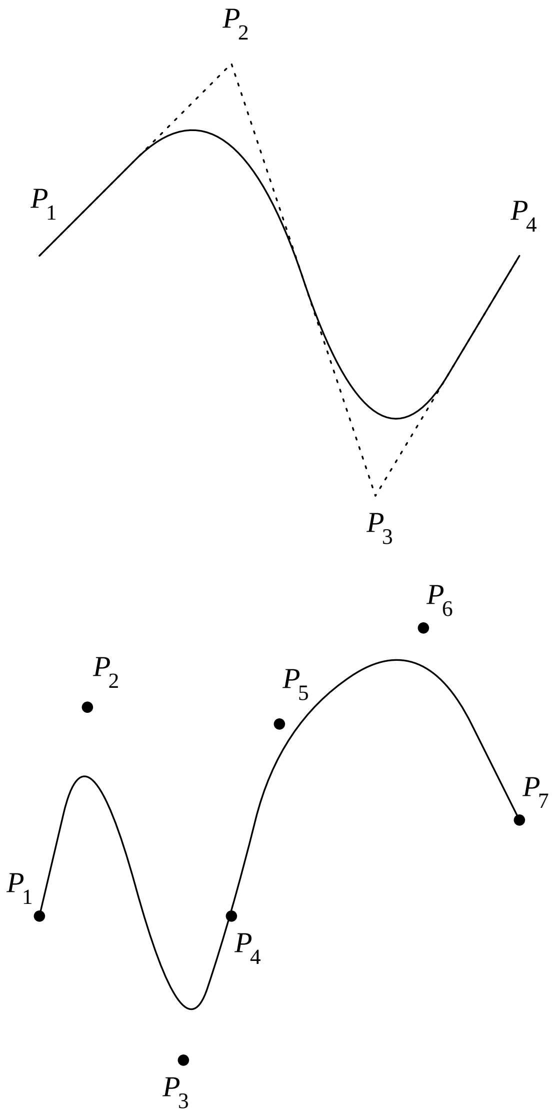

Figure 2

3 METAFONT to POSTSCRIPT compiler—why and how

This section describes a major performance problem with METAFONT-generated fonts that perhaps can be solved by translating them into POSTSCRIPT fonts. The goals of this translation are established. Based on these goals, a particular approach is adopted to engineer the software largely from existing components.

3.1 A problem with METAFONT-generated bitmapped fonts. In METAFONT, one gets one bitmap per point size and magnification. The size of these bitmaps grows as the square of product of the design size and the magnification and requires a large storage space. Files that are sent to the printer will be large, especially if lots of different point sizes or magnifications are used. In POSTSCRIPT with outline fonts, there is one outline per character which can be scaled arbitrarily to any magnification that might be needed. Moreover, POSTSCRIPT outline fonts are generally more compact than bitmapped fonts. For example, an enclosed rectangle is represented by its four corner points rather than by all the bits enclosed by the rectangle.

Certainly the outline fonts are more easily scaled to other magnifications. By scaling the bitmapped fonts downward, too much information is lost, and scaling upward introduces the jaggies. Moreover, the pixel array is device dependent; it is valid for output devices of only one particular resolution and one choice of possible data values per pixel. Scaleable fonts have a great advantage — you need only one font description file for all magnifications of that font. Actually, POSTSCRIPT outline fonts are more scaleable even than the METAFONT originals for another reason. In [9], it is said, "Caution: before using this '**at**' feature (i.e. scaling downward or upward) you should check to make sure that your typesetter supports the font at the size in question; TEX will accept any ⟨desired size⟩ that is positive and less than 2048 points, but the final output will not be right unless the scaled font really is available on your printing device." Getting POSTSCRIPT outline versions of METAFONT fonts is possible since both are based on Bézier curves. Doing so makes fonts heretofore available only on TEX and other DVI-based formatters available on ditroff and other formatters which have evolved to or have been designed for use with POSTSCRIPT printers.

3.2 Goals. Based on the observations of Section 3.1, the goal of this research is to produce a METAFONT to POSTSCRIPT compiler, mf2ps. Its operational requirements are items 1 through 5:

1. It must be possible to translate any legitimate METAFONT font definition at any given design size into a POSTSCRIPT outline font.
2. The resulting POSTSCRIPT outline font should be arbitrarily scaleable.
3. The resulting fonts should look like the bitmapped fonts when printed on the same printer.
4. The resulting POSTSCRIPT outline font should be more compact *when sent to the printer* than a POSTSCRIPT version of the METAFONT-generated bitmapped font.

The fourth requirement deserves a bit of explanation and qualification. First note that what is compared is what is sent to the printer. Certainly there are compressed versions of the bitmapped fonts that reduce the disk storage requirements of the bitmapped fonts. However, they must be uncompressed before sending them to most printers. It is the printer's storage that is limited; generally disk space is in abundance. However, since printers these days are general purpose computers, what a printer accepts may in fact be a compression that it has been programmed to undo.

Now for the case in which disk space is of concern, the comparison should still be relative to printable versions. There exist algorithms, e.g. that of Lempel and Ziv [13] that can be used to compress POSTSCRIPT outline fonts which are, after all, just ASCII files. Therefore, in order not to have a contest between compression algorithms, the uncompressed versions are compared. Furthermore, in order not to have a contest between different kinds of printers that may have differing font representations, POSTSCRIPT outline fonts are compared to POSTSCRIPT bitmapped fonts. When considering disk space, the fact that one bitmapped font is needed for each magnification is taken into account. Thus, the interest is in comparing the size of a scaleable outline font to the total storage for the bitmapped fonts for all magnifications of a given design size.

5. The resulting POSTSCRIPT outline font should be more compact than the total of the sizes of the POSTSCRIPT versions of the METAFONT-generated bitmapped fonts at each available magnification. Even this comparison is not completely fair since only specific magnifications are provided, while the POSTSCRIPT font is arbitrarily scaleable.

Observe finally, that the comparison is against magnifications of a single design size since purists would argue that there should be a different outline font for each design size. Since there are those that do not require this purity, the various design sizes will be compared also.

The software engineering goal is item 6.

6. mf2ps should be written as much as possible using the existing METAFONT program both to save work and to ensure that all METAFONT-acceptable font definitions are handled.

The evaluation of the results will be done relative to these goals.

3.3 Previous attempts. Leslie Carr wrote a collection of programs to produce POSTSCRIPT outline fonts from METAFONT fonts in 1987. Carr's programs take as input the *log* output file of METAFONT which contains a description of all the paths that METAFONT traces out in drawing a character.

Carr has problems of information loss as a result of not having entered into the METAFONT program. This is the reason why Carr's characters are poor looking. In [5], Carr observes, "In the `cmr10` font, the *crisp* pen has diameter zero, so serifs have square corners. In the `cmtt10` font, *crisp* is set to a larger value and the serifs end in semicircles. Because the shape of the current pen can NOT be taken into account in POSTSCRIPT, these differences in the characters shapes will not be seen. This is a **fundamental** problem: given a path p and a pen q (whose shape is also an arbitrary path), METAFONT effectively envelopes p with respect to the shape of q; POSTSCRIPT can do nothing other than stroke it to produce a line of constant width. This incompatibility comes to light when the width of the pen is significant to the shape of the character".

In order to avoid this problem, mf2ps finds the internally generated envelope, which is used as the boundaries of the inked region, and uses this envelope as the outline. It does not matter, then, what the pen path and the pen shape are.

More recently, during the time that the work described herein was being done, there were other efforts with similar goals.

Doug Henderson [6] obtained outline font characters by modifying the `endchar` macro, which is called for each character after the bitmap is generated, to take the bitmap for the character and white out all but the bits on the edge. The number of bits left on the edge is varied according to the resolution of the bitmap. These outlines, being bitmapped, are just as unscaleable as are the bitmaps for the filled-in characters.

Neil Raine and Graham Toal [12] have developed software that takes the bitmaps and rediscovers the outlines by tracing the pixels. The outlines that are used as the basis for POSTSCRIPT fonts are, for the most part, generated from bitmaps at 2400 dpi. They first generate RISC OS outline fonts which are screen fonts for Acorn's Archimedes RISC computer. These are true scaleable outlines. Then, these outlines are converted into POSTSCRIPT format. Toal says that the the quality of the fonts produced is not too great at low resolutions because of shortcomings in Adobe's rendering algorithm. He adds that at 1200 dpi on a phototypesetter, they are indistinguishable from METAFONT-generated bitmapped fonts. These authors suspect that information that is critical for good appearance is lost when tracing an outline on a bitmap generated from a mathematically described envelope. Better results should be obtainable using the original envelope.

John Hobby [7] has developed a program called MetaPost, which translates from an extension of METAFONT into POSTSCRIPT cubic splines and commands. His goal was to turn METAFONT into a system for typesetting general graphics, including embedded text. His approach, similar to ours, was to modify the METAFONT program into what he desired. Befitting his more general goals, besides modifying the output, he has added new commands to the input language. Moreover, his translation appears to be a direct mapping from a METAFONT command sequence to a POSTSCRIPT command sequence. The result is a program more powerful than mf2ps. It will be interesting to compare fonts produced by MetaPost and mf2ps for appearance and performance.

3.4 Methodology. There are a number of ways to build the compiler. They include

1. writing the whole compiler from METAFONT to POSTSCRIPT from scratch: This has the advantage that one does not have to get into another person's software, which is not very pleasant when the software is so big. On the other hand, one would have to treat the whole job of turning mathematical equations and any arbitrary pen shape into outlines.

2. using the METAFONT output as was done by Leslie Carr [5]: This has the advantage of not requiring delving into another's software, but the generated information is not enough if one wants no deviations from the originals.

3. getting into the METAFONT program: This requires examining the internals of the METAFONT program. However, METAFONT and POSTSCRIPT make liberal use of Bézier curves to describe non-circular curves. This fact makes the translation process natural. For each specified path, METAFONT creates control points for the associated cubic spline curves before calculating the bit map. METAFONT also calculates the edge offsets implied by the pen shape. Using the necessary information one can get a new set of control points that define Bézier curves and lines that are needed to build the POSTSCRIPT outline fonts.

3.5 Software engineering of solution. The idea is to split the METAFONT program into front end and back end. The front end takes METAFONT specification of a character, magnification, and point size, and produces the envelope, i.e., the outline of the character, and the back end fills the envelope with bits. Taking the existing front end and writing a new back end that converts the envelope into a POSTSCRIPT specification of an outline is our method of producing mf2ps. The bit-filling process will be done by the printer.

In order to make POSTSCRIPT fonts arbitrarily scaleable, we have to ask the mf2ps program to use a very large magnification, at least to try to match the grid on which Adobe plots the points of its outlines. Adobe plots its characters on a 1000×1000 grid. Thus, Adobe's resolution is 1000 dpm (dots per em), which for design size 10 points is 7200 dpi. Unfortunately, METAFONT, and thus mf2ps accepts resolutions only up to 3000 dpi. The results should be sufficient to produce fonts scaleable up to magnification 7 or 8, which is a reasonable range in typesetting.

This approach helps meet goal 6 because the original unchanged METAFONT program is used. Thus, exactly the same input is accepted as in the METAFONT program. There is some extra frosting obtained by the chosen approach. The program for translating METAFONT to POSTSCRIPT is actually a bit of an interactive environment because the new back end is an extension of the existing one. This existing back-end provides an interpreter that executes a METAFONT character definition and displays the defined character on the screen. Figure 3 shows the dump of a screen containing several windows, one showing a METAFONT definition, another showing the result of its interpretation, and a third containing the POSTSCRIPT translation of the definition in the first window. If software to interpret POSTSCRIPT definitions were available here, a fourth window could be set up showing the result of interpreting the translation of the third window. This would allow comparison of the character's appearances without having to print them on paper.

4 The program

In the following discussion, the METAFONT program is often called just "METAFONT".

The METAFONT program has been written so that it can be made to run efficiently in a wide variety of operating environments by making comparatively few changes. Such flexibility is possible because the program is written in the WEB language which is at a higher level than Pascal. The preprocessing step that converts WEB to Pascal is able to introduce most of the necessary refinements. Semiautomatic translation to other languages is also feasible, because the program does not make extensive use of features that are peculiar to Pascal.

The program has two important variations: First, there is a long and slow version called INIMF, which does the extra calculations needed to initialize METAFONT's internal tables. It has to be run first. It initializes everything from scratch without reading a base file, and it has the capability of dumping a base file. Secondly, there is a shorter and faster production version called VIRMF, which cuts the initialization to a bare minimum. It is a virgin program that needs to input a base file in order to get started. VIRMF typically has more memory capacity than INIMF, because it does not need the space consumed by the dumping and undumping routines, etc.

In order to generate a compiler that translates METAFONT to POSTSCRIPT, additional external procedures and functions were added to the METAFONT program so that it runs exactly the same except that when it asks for an output file name, it asks for an additional name, for the extra output file that is to contain the POSTSCRIPT outlines. Those changes were made on the Pascal version of the VIRMF, and were compiled later with METAFONT's library files. (It was a complete oversight on our part not to have modified the WEB version of VIRMF.) A few extra lines were added to the macro file, `plain.mf`. These act as flags, identifying that METAFONT has entered some of the macros.

4.1 Basic idea. To specify a character in METAFONT, one specifies either an envelope (outline) or a center-line path and a pen head. For the former, METAFONT just fills the envelope with bits. For the latter, METAFONT pretends that it is drawing the character with a pen of specified head shape following the specified path, i.e., the center of the head stays on the path. The distance from the center-line path and outer edge of ink trail left by pen head is called the *offset*. So, for a character, METAFONT follows the center-line path to calculate the path of offset points, i.e., the envelope, and then fills the envelope with bits. In either case, METAFONT ends up filling an envelope.

We need to break METAFONT into a front end and a back end at the point just after the envelope has been calculated. Then we provide a new back end that converts the envelope into POSTSCRIPT instead of filling the envelope with bits. Note then that the POSTSCRIPT printer will fill in the envelope with bits as it fills the path obtained from the envelope.

The following subsections describe the data and the calculations involved in the new back end.

4.2 Data structures. The main data structures that METAFONT keeps for a character are the center-line path, the pen shape, and the envelope path. There are a few operations that can be performed on paths, called transformations.

4.2.1 METAFONT's path representation. When a METAFONT user specifies a path, METAFONT creates a list of knots and control points for the associated cubic spline curves. If the knots are $z_0, z_1, \ldots, z_n$, there are control points z_k^+ and z_{k+1}^- such that the cubic splines between the knots z_k and z_{k+1} are defined by the Bézier formula

$$\begin{aligned} z(t) &= B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t) \\ &= (1-t)^3 z_k + 3t(t-1)^2 z_k^+ \\ &\qquad + 3t^2(1-t) z_{k+1}^- + t^3 z_{k+1}, \end{aligned}$$

for $0 \le t \le 1$.

There is a 7-word node for each knot z_k, containing one word of control information and six words for the x and y coordinates of z_k^- and z_k and z_k^+. The control information appears in the *left_type* and *right_type* fields and they specify properties of the curve as it enters and leaves the knot. There is also a *link* field, which points to the following knot. Before the Bézier control points have been calculated, the memory space they will ultimately occupy is taken up by information that can be used to compute them. The METAFONT *make_choices* procedure chooses angles and control points for the splines of a curve when the user has not specified them explicitly.

4.2.2 METAFONT's path transformation. When METAFONT digitizes a path, it reduces the problem to the special case of paths that travel in the *first octant* directions; i.e., each cubic $z(t) = (x(t), y(t))$ being digitized will have the property that $0 \le y'(t) \le x'(t)$. This assumption makes digitizing simpler and faster than if the direction of motion has to be tested repeatedly. When $z(t)$ is cubic, $x'(t)$ and $y'(t)$ are quadratic, hence each of the four polynomials, $x'(t)$, $y'(t)$, $x'(t)-y'(t)$, and $x'(t)+y'(t)$, crosses through 0 at most twice. If we subdivide the given cubic at these places, we get at most nine subintervals. In each of these intervals each of $x'(t)$, $y'(t)$, $x'(t)-y'(t)$, and $x'(t)+y'(t)$ has a constant sign. The curve can be transformed in each of these subintervals so that it travels entirely in first octant directions, if we exchange x and $-x$, y and $-y$, and x and y as necessary.

4.3 Pens and envelopes. There are two kinds of pen heads that may be used, polygonal and elliptic. There are a number of trade-offs involved in their use. The first subsection treats the case of an n-vertex polygonal pen shape and the second treats the case of an elliptical pen shape. Both describe the influence of pen shape on the envelope of the font.

4.3.1 Polygonal pens. Suppose that the vertices of a polygon are $w_0, w_1, \ldots, w_{n-1}, w_n = w_0$ in counterclockwise order. A convexity condition requires that each vertex turns left when one proceeds from w_0 to $w_1 \cdots$ to w_n. The envelope is obtained if we offset a given curve $z(t)$ by w_k when that curve is traveling in a direction $z'(t)$ lying between the directions $w_k - w_{k-1}$ and $w_{k+1} - w_k$. At times t when the curve direction $z'(t)$ increases past $w_{k+1} - w_k$, METAFONT temporarily stops plotting the offset curve and inserts a straight line from $z(t) + w_k$ to $z(t) + w_{k+1}$; notice that this straight line is tangent to the offset curve. Similarly, when the curve direction decreases past $w_k - w_{k-1}$, METAFONT stops plotting and inserts a straight line from $z(t) + w_k$ to $z(t) + w_{k-1}$; the latter line is actually a retrograde step, which will not be part of the final envelope under METAFONT's assumptions. The result of this consideration is a continuous path that consists of alternating curves and straight line segments. The segments are usually so short, in practice, that they blend with the curves.

4.3.2 Elliptical pens. To get the envelope of a cyclic path with respect to an ellipse, METAFONT calculates the envelope with respect to a polygonal approximation to the ellipse. This has two important advantages over trying to obtain the exact envelope:

1. Polygonal envelopes give better results, because the polygon has been designed to counteract problems that arise from digitization; the polygon includes sub-pixel corrections to an exact ellipse that make the results essentially independent of where the path falls on the raster.
2. Polygonal envelopes of cubic splines are cubic splines. Hence it is not necessary to introduce completely different routines. By contrast, exact envelopes of cubic splines with respect to ellipses are complicated curves, more difficult to plot than cubics.

4.4 Taking out data. After METAFONT has calculated the paths and the offsets, it is ready to send the values to the *make_moves* procedure which generates discrete moves for any four points that represent a Bézier curve. This is done for each one of the cyclic paths from which the letter is built. When the offsets are zero, this is done by the *fill_spec* procedure. Otherwise this is done by the *fill_envelope* procedure. In the latter case, the line segments, which were discussed earlier, should be taken out also in order to get smooth connections between the different curves that the cyclic path is built from. Because POSTSCRIPT describes any shape in terms of curves and lines, this is the point to take advantage of METAFONT's calculations, i.e., when METAFONT calls the *make_moves* procedure and when METAFONT draws line segments for offset corrections.

4.5 Processing the data. The generated data are not ready yet to be used. First, we should unskew, i.e., transform from the first octant back to the original, the paths according to the octant that the paths were traveled in before they were skewed. This unskewing is done by taking out the octant number at the moment that the *make_moves* procedure is called and then using METAFONT's *unskew* procedure that sets values x' and y' to the original coordinate values of a point, given an octant code and coordinates (x,y) after they have been mapped into the first octant and skewed; the new values are sent to the *send_p_s* procedure. This procedure has eight formal parameters that are all used when sending a curve. When sending a line, only four parameters are used, two to denote the start point and two to denote the end point; the remaining four parameters are sent as zeros so *send_p_s* can distinguish whether a line was sent or a curve. In the next step, *send_p_s* unscales the numbers because METAFONT works with units of scaled points, of which there are 2^{16} in an ordinary point. While unscaling, the values are transformed in order to send them to the POSTSCRIPT dictionary `FontBBox` command. After this pre-processing, the data are sent to a temporary file.

4.5.1 Getting more information. When METAFONT calls the *make_moves* procedure, it does not have any information on the role that this path is going to play, whether the current cyclic path is going to be *filled* or whether it will act as a boundary of a region to be *erased*.

In order to distinguish between the cases, more information has to be taken. This is done by copying the `plain.mf` file into a new file named `myplain.mf` and adding a few lines to it. The additional code was added in order to identify METAFONT's use of the macros. METAFONT uses the variables for date only once, when the program is started, so it was decided to use them in the rest of the program. The `year` is changed to –1 when METAFONT's `pen_stroke` macro is applied on a cyclic path, i.e., in the characters such as "o", "O", and "Q", and to –2 when the `erase` macro is called. The `month` is changed when the `fill` macro is called. There are three kinds of paths:

1. paths to be *filled* are processed using the POSTSCRIPT `fill` command.
2. paths to be *stroked* are processed using the POSTSCRIPT `eofill` command.
3. paths to be *erased* are processed using specialized procedures which will be discussed later.

A letter cannot always be treated as one unit by means of the `fill` and `eofill` commands. For instance, the letter "Q" is built of two different paths, the first of which is stroked and the second of which is filled. Generating the letter using the POSTSCRIPT `eofill` command causes a hole in the image (see Figure 4).

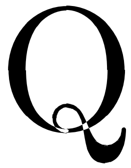

Figure 4

So while generating a letter, fill mode can be changed for each cyclic path. Moreover, when generating a letter whose paths should be filled, it is not always possible to use just one `fill` command (see Figure 5).

Figure 5

When a POSTSCRIPT `fill` command is applied to a path that is composed of more than one subpath, say two for the sake of simplicity, and one subpath is inside the other and is drawn in a direction opposite to the external one, the internal path is considered a hole and is not filled (see Figure 6). So, if several paths are to be filled in this manner, the description of each one of them should be ended with the `fill` command. There is one more benefit to using this strategy: The POSTSCRIPT `current path` stack becomes empty after encountering any kind of `fill` command. Therefore, using the `fill` command after each path can help avoid `stack overflow error`s if all paths together are too long.

4.5.2 Treating erasing paths. There are three methods of handling the problem of paths that should be erased by mf2ps itself:

1. filling with white: Because erasing paths are built in order to erase an existing filled area and POSTSCRIPT overlaps paths (i.e., a region is shown in the color that was drawn last), erasing paths can be implemented by filling those paths with white. This solution is the easiest, but it works only if the background is white and the letter is drawn in some level of gray. If one wants to draw a letter with background other than white, the resulting

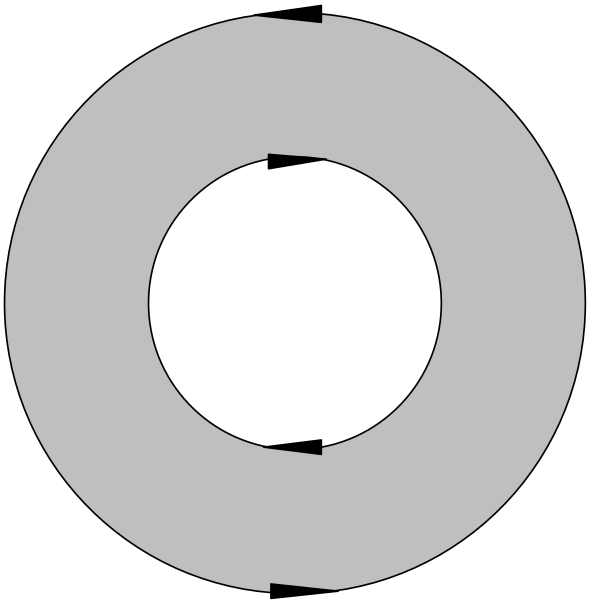

Figure 6

appearance will not be correct.

2. calculating new paths resulting from subtracting the erasing paths from the previous filled paths: Such a solution can be global. However, it costs a lot in terms of processing time and accuracy, because paths are given implicitly by four points, and in order to calculate the new paths, one should find the intersection points of Bézier curves, i.e., to find points that lie on both Bézier curves, and then calculate new curves, which are difficult to calculate from those points.

3. using the POSTSCRIPT `eoclip` command: Because the letters are bounded in a 1000 $\times$1000 box, a primary square path whose segments are 1000 units long should be declared and after it all the erasing paths should be listed. After relocating the erasing paths we are ready to declare `eoclip`, which means that the clipping path is the external primary one and the internal paths, the erasing paths, are holes. This is an elegant solution that uses the power of the language and is available in simple situations in which there is no intersection between the erasing paths (see Figure 7). If there were intersections, a little more sophisticated use of the `eoclip` command would be needed. Relocation of the erasing paths is done by the procedure `doarrange`.

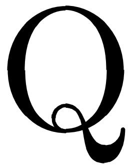

Figure 7

There are other problems caused by the erasing paths. Because the erasing paths have segments in common with paths to be filled, POSTSCRIPT must decide whether the common segments are in the clipping path or not. POSTSCRIPT does not seem to have a consistent policy on that and it seems to be that the decision is taken arbitrarily (see Figure 8).

Figure 8

An attempt to resolve the clipping path problem led to the first author sending the following electronic message (obviously, not as nicely formatted as herein) to Glenn Reid of Adobe Systems, Inc.

> From simon Tue Mar 21 13:22:32 1989
> To: greid@adobe.com
> Subject: Problem in PostScript
>
> Dear Mr. Reid
>
> I have got a problem in understanding the PostScript policy in determining "what is in the clipping path". I think there is a problem in the boundaries. Here is an example that shows that problem:

```
gsave
initclip
newpath

0 0 moveto
0 1000 lineto
1000 1000 lineto
1000 0 lineto
0 0 lineto

300 100 moveto
700 100 lineto
700 300 lineto
300 300 lineto
```

```
300 100 lineto

700 900 moveto
300 900 lineto
300 700 lineto
700 700 lineto
700 900 lineto

eoclip

newpath
100 100 moveto
900 100 lineto
900 900 lineto
100 900 lineto
100 100 lineto
fill
grestore
```

> As you see, the problem is that on top of the shape, the line which belongs to the upper "hole" in the clipping path and to the current path (to be filled) is drawn, and on bottom of the shape it is not.
>
> This is happening both on the Apple Laser printer and on the QMS-80.
>
> I would be glad to have a reply from you.
>
> Thanks in advance
> Shimon Yanai
> C.S Dep.
> Technion

What Mr. Reid saw when he printed the POSTSCRIPT commands contained in the message is reproduced in Figure 9.

Figure 9

Mr. Reid replied with the following:

> From: greid@adobe.com (Glenn Reid)
> To: Shimon Yanai <simon@techunix>
> Cc: greid@adobe.com
> Subject: Re: Problem in PostScript
> In-Reply-To: Your message of Wed, 22 Mar 89 ...
> Date: Wed, 22 Mar 89 11:41:35 PST
>
> The problem is that the path you are filling falls exactly on the edge of the clipping path. This produces a zero-width area to fill, and unfortunately it sometimes fills and sometimes does not with the current fill algorithm. I believe that it is related to the direction of the paths; if the paths are going in opposite directions along the same line, it will fill with a one-pixel area, but if they are going in the same direction, it will not fill. I believe this has been fixed to be more consistent in Display PostScript, for what it's worth.
>
> Glenn Reid
> Adobe Systems

The idea of using opposite directions had been checked before sending the letter, so the problem had to be solved within the back end of mf2ps. The erasing paths near the top of the letter had their *y* coordinates increased by 0.8 points, and those near the bottom had their *y* coordinates decreased by the same amount. This shift is invisible to the human eye because the font definitions are in terms of hundreds of points (see Figure 10). This solution was designed to work with most existing METAFONT fonts. It is possible that there will be fonts that are not treated well by this solution.

Figure 10

4.6 Optimization. Optimization is done in order to make the description of the fonts shorter and to save work in the POSTSCRIPT interpreter. This is done in three ways:

1. not printing lines with length zero. As was said earlier, the METAFONT program prints lines to connect offset points. There are times that after rounding or truncating the output data, the start point and the end point are equal. In such cases, the lines are eliminated.
2. checking if the Bézier curve acts as a line. From the definition of the Bézier curve, it is known that if the two control points lie on the line that connects the start point and the end point, the curve is of degree one. In such cases mf2ps generates a command to print a line from the start point to the end point, thus saving space and avoiding redundant calculations for the POSTSCRIPT interpreter.

3. checking if a series of consecutive line segments are in the same line. This is done by storing the segments in a buffer and checking whether a new segment is collinear with the last stored.

4.7 Changed or added routines. The following is a list of routines that were changed or added in order to build mf2ps from METAFONT.

printchar was modified to get character names.

fixdateandtime was modified to initialize variables that were used as flags in the macros.

fillspec was modified to send out data on splines.

skewlineedges was modified to send out offset lines.

dualmoves was modified to send out offset lines.

fillenvelope was modified to send out data on splines.

dostatement was modified to identify tokens that are strings.

main was modified to call the mf2ps procedure in the beginning and ending of the program.

sendcurve was added to unskew spline values and to send them to the next process.

sendline was added to unskew line values and to send them to the next process.

ok was added to check if two lines are collinear.

restore was added to restore the parameters of the last line.

recall was added to recall values from the buffer.

us was added to convert the METAFONT scale so that a letter would fit the Adobe standard 1000×1000 bounding box.

send_p_s was added to create a POSTSCRIPT file of lines and curves.

makemoves was modified to send out spline data.

dump was added to append information from the file named `f` to the file named `g`.

checkerase was added to identify the file that contains "erase" commands, and their position within the file.

doarrange was added to put erasing paths at the beginning of the file.

print_start was added to signal the beginning of a new cyclic path to be processed.

print_end was added to signal the end of the current cyclic path.

init_ps was added to make initializations.

makenewdef was added to make initializations when more than one character occurs in the input.

closeolddef was added to close the last definition.

tini_ps was added to handle the ending of the process.

auxprintchar was added to print characters.

auxprint was added to print strings.

5 Operation of mf2ps in a UNIX environment

When invoked, mf2ps first asks for an output file name. For the example this file is called `ex1`. mf2ps then asks,

> `"Are you creating the whole dictionary (y/n)?".`

If the answer is other than "`y`" or "`Y`", it is considered "`no`". If the answer is "`y`" or "`Y`", then the whole dictionary is created. This means that mf2ps creates a POSTSCRIPT dictionary that includes entries for all the characters that are in the input, e.g., `cmr10` set. This dictionary needs additional definitions such as *left side bearing, width, bounding box,* etc. These definitions need information on character features that must be calculated within the program. Otherwise, the whole dictionary is not created and the program treats the input as a single character definition that is to be translated into a POSTSCRIPT outline definition. After mf2ps prompts "`***`", we are in the METAFONT environment. Now the user inputs

```
\mode=hires;\nodisplays;\input cmr10;↲
```

After mf2ps has finished, the resulting POSTSCRIPT font dictionary can be used to print text. In order to print text, the font dictionary should be installed in some formatter's font source directory, and then it can be loaded through the formatter's commands. The dictionary followed by appropriate show and showpage commands can also be sent directly to the printer.

6 Evaluation of results

This section evaluates the mf2ps program relative to goals established in section 3.2. The program was produced as a variation of METAFONT and it accepts any METAFONT font definition and produces a POSTSCRIPT outline font scaleable up to magnification 8, or to point size 80 if you are not a purist. Thus goals 6 and 1 have been entirely met and goal 2 is partially met. To meet goal 2 fully the program must be modified to allow large enough arrays to handle magnifications up to 7200; this is left to future work.

It remains to evaluate the appearance and sizes of the outline fonts relative to the bitmapped fonts to see if goals 3, 4, and 5 have been met.

6.1 Appearance. In order to compare appearances, the outline font (Subsubfigure P) and and the 300 dpi bitmapped font (Subsubfigure M) generated from the same METAFONT definition are used to print similar sentences at one, two, or three different sizes or magnifications on three devices of differing resolutions. The sentences are printed in the `cmr` (Subfigure R), `cmtt` (Subfigure T), and `lasy` (Subfigure S) typefaces. The bitmapped fonts may be printed at design sizes 7, 8, 10, or 12, and the outline fonts may be printed at magnifications .7, .8, 1.0, or 1.2. Finally, the three devices are the 300 dpi LaserWriterII (Figure 11-LW300), the 600 dpi Varityper (Figure 11-VT600), and the 1270 dpi Linotronic 300 (Figure 11-LT1270). The bitmapped font examples are formatted with TEX while the outline font examples are hand-coded POSTSCRIPT files sent directly to the printer. Since the formatter with which this paper is printed can use arbitrary POSTSCRIPT fonts, half of the examples could have been done in-line without pasting in. However, for fairness in the comparison, all examples were cut out and pasted in.

There are visible differences due to differences in the formatting software. TEX squeezes the letters closer together than does the POSTSCRIPT engine. Moreover, the interword space is constant in the POSTSCRIPT dictionary but is varied by TEX according to the line structure. These differences are not the differences that are at issue here.

On the 300 dpi device, the characters from the bitmapped fonts print thinner than are those of the outline fonts. However, the edges of both sets are equally smooth or jagged as the case may be in all sizes. Overall, then, the appearance of the characters of the bitmapped fonts is crisper than that of the outline fonts. On the higher resolution devices, the thicknesses of the characters are closer to being equal at all sizes. Thus, the METAFONT program does a better job of building a correctly sized bitmap at 300 dpi than does the 300 dpi POSTSCRIPT engine of the LaserWriterII. The latter seems to round up too much. However, both seem to get the edges equally smooth even at low sizes and low resolutions.

At the two higher resolutions, the outline fonts are significantly better than the outline fonts at lower resolutions and are significantly better than the bitmapped fonts at the same resolution of printing. However, this latter is true because the bitmapped fonts were generated by the METAFONT program specifically to be printed at 300 dpi. When a 300-dpi bitmap is printed with no scaling at 600 or 1270 dpi, it remains a 300-dpi bitmap. As expected, the 300-dpi bitmapped fonts print better at 300 dpi than they do at the two higher resolutions.

The generated outlines are not fine-tuned for printing at low resolutions, such as 300 dpi, as are the METAFONT-generated bitmaps. It might be useful to make use of the POSTSCRIPT facilities for hinting to improve the appearance of the characters printed from the outlines at low resolutions.

Figure 12 shows samples of similar sentences printed on the same three devices using the standard Helvetica, Times Roman, and Courier POSTSCRIPT outline fonts built into most POSTSCRIPT-executing laser printers. It appears to these authors that the standard POSTSCRIPT fonts are significantly better than those generated from METAFONT fonts. However, this is not surprising. Adobe uses a grid of 1000×1000 for its character definitions, resulting in a resolution of 7200 dpi for characters printed at point size 10. Because of size limitations of the METAFONT program the METAFONT outline fonts are using a resolution of 3,000 points per inch. However, when using the letters in small sizes such as from 10 to 70, quality differences are hardly visible especially when working with printers that have a resolution of 300 points per inch such as the Apple LaserWriter. Moreover, Adobe makes liberal use of hinting to improve the appearance of its fonts at low resolutions. We completely ignored hinting, as we did not see any way to automatically generate the hints.

6.2 Sizes of fonts. Recall that it is necessary to compare the size of the POSTSCRIPT outline font for a particular METAFONT definition to the sizes of the bitmapped fonts in POSTSCRIPT fonts for the individual and all magnifications.

This comparison is made in this section for the `cmr10` font at the standard set of six magnifications 1, 1.095, 1.2, 1.44, 1.728, and 2.07 (which are approximations of 1.2 raised to the powers 0, .5, 1, 2, 3, and 4, respectively). In addition, as a gesture to those who are not purists and accept magnifications of the 10 point design size as different point sizes, the comparison includes the `cmr` font at point size 5, 6, 7, 8, 9, 10, 12, and 17, the standard eight design sizes maintained for use with TEX.

Table 1 shows the sizes in bytes. Thus it is clear that the POSTSCRIPT outline font is bigger than any bitmapped font and that goal 4 fails. Moreover, it is clear that the outline font is bigger than the sum over all magnifications of one design size and than the sum over all standard design sizes. Thus goal 5 fails. In fact, this failure is the reason that the samples of Figure 11 involve only upper case letters. Samples with complete fonts with both cases often overloaded the printer available to the students at the time this work was done.

Font	Design size	Magni-fication	Bitmap (size in bytes)	Outlines (size in bytes)
cmr	10	1.0	22,812	245,000
"	10	1.095	24,231	"
"	10	1.2	26,044	"
"	10	1.44	31,892	"
"	10	1.728	39,614	"
"	10	2.07	50,578	"
cmr	5	1.0	16,729	"
"	6	1.0	17,757	"
"	7	1.0	18,820	"
"	8	1.0	20,041	"
"	9	1.0	21,580	"
"	12	1.0	25,658	"
"	17	1.0	37,140	"
Total			352,896	245,000

Table 1

However, do note that the outline font is smaller than the sum over all design sizes and magnifications thereof.

So in terms of disk space for the non-purists, the outline font represents a savings. Again notice that not all magnifications of the bitmapped fonts are maintained and the outline font is arbitrarily scaleable. Moreover, as the magnification grows the size of the bitmap grows even more rapidly.

The disappointment with respect to saving printer and disk memory says that it is important to spend more effort to optimize the outline font.

All is not lost, though! As this paper was being prepared for publication in *TUGboat*, one reviewer, Nelson Beebe, pointed out something that we can only kick ourselves for not noticing. The POSTSCRIPT outline fonts that are generated by mf2ps are horrendously wasteful in space. They use original, built-in command names and absolute coordinates. A significant reduction in size can be obtained by definition and use in the outlines of single-character command names, e.g., “M” for “moveto”, and by use of relative versions of these commands with operands of fewer digits after the initial absolute moveto of any character. A simple filter was written to obtain new compressed versions of the POSTSCRIPT outline fonts. The appearances of the output when printing with these new versions is unchanged, but what is sent to the printer is significantly smaller, about 37.7% smaller. The reduction on a per-letter basis is about 45%. Table 2 shows the information of Table 1 for the new versions of the outline fonts.

Font	Design size	Magni-fication	Bitmap (size in bytes)	Outlines (size in bytes)
cmr	10	1.0	22,812	152,670
"	10	1.095	24,231	"
"	10	1.2	26,044	"
"	10	1.44	31,892	"
"	10	1.728	39,614	"
"	10	2.07	50,578	"
cmr	5	1.0	16,729	"
"	6	1.0	17,757	"
"	7	1.0	18,820	"
"	8	1.0	20,041	"
"	9	1.0	21,580	"
"	12	1.0	25,658	"
"	17	1.0	37,140	"
Total			352,896	152,670

Table 2

There are still better compressions that can be achieved. According to Beebe [4], Toal and Raine’s outline representation of cmr at 10 points requires about twice the space needed for bitmaps of the same; at 14 to 16 points, the outlines and the bitmaps occupy about the same amount of space; above 16 points, the outlines are smaller than the bitmaps. It is clear that better encodings exist than we explored and these must be explored for any future version of mf2ps.

One such better encoding appears to be that used by Adobe for its own proprietary fonts; fonts encoded this way have a FontType of 1. User defined fonts have a FontType of 3. Beebe [4] says that type 1 fonts are handled with greater efficiency than type 3 fonts on most existing POSTSCRIPT interpreters, especially those that are based on Adobe-licensed code. Adobe has recently published specifications for the type 1 font encoding [2], thus allowing anyone to produced type 1 fonts. Beebe believes that the market forces will drive other companies to encode their fonts as type 1. Moreover, as more and more windowing systems based on POSTSCRIPT, e.g., NeWS and NeXT, appear, the attraction of POSTSCRIPT outline fonts will increase, as then the same font can be used for both printing and previewing. Thus, the incentive will be to convert META–

FONT fonts into type 1 POSTSCRIPT outline fonts.

Ultimately, the tradeoff is between the size of the font sent to the printer, and the time it takes for the printer to decode the program for the characters. However, with proper cacheing, a big enough cache, and a not very fancy document, the decoding is done only once per character for the document!

7 Future work

For the future, there are a number of improvements that can be made. Currently, each letter of the POSTSCRIPT outline fonts is described as a set of cyclic paths. When all are filled or stroked, one gets the desired letter. Some of those cyclic paths have a common boundary that is inside the letter and is not necessary for the outline description of the letter as a whole. Eliminating these paths and creating one outline for the letter will save space. Today this can be done manually, and is worth the effort because the translation process is done only once. From that time on, the font is used the way it is.

As was demonstrated by Beebe's rescue of our result, closer attention should be paid to obtaining more compact representations of character outlines, representations for which POSTSCRIPT routines can be written to interpret them into standard outline drawing commands. Collapsing commands into single characters and using relative movements saved significant amounts of space. Perhaps, even more dramatic savings can be obtained by giving coordinates and distances in hexadecimal.

More effort can be spent on modifying the program in order to allow magnifications up to 7200 points. Thus, no jaggies will be seen, as occasionally happens when using higher magnifications, e.g., in our translated fonts at magnification 8. This could be done by enlarging the program arrays to handle characters based on 7200 points. A sophisticated solution is required if one wants to save room while compiling the input font. In such a case, any linear translation which is done within the POSTSCRIPT program is with a factor less than 1.

METAFONT was changed for TEX 3.0. It is necessary to build a new version of mf2ps based on this latest version of METAFONT. As the changes to the METAFONT program deal mainly with ligatures and kerning, the calculation of envelopes is probably not affected. Therefore, it is likely that the portion of META–FONT up to the calculation of the envelope can still be used as a front end for mf2ps with very little change in the portion of the program we wrote.

Finally, it might be worthwhile, for the sake of portability to other systems and enhanceability by other humans, to rewrite or to write the next version of mf2ps with WEB.

Acknowledgments

The authors thank the *TUGboat* editors, and Nelson Beebe for their help, sharp comments, and result-saving ideas. Dealing with their comments made this a better paper.

References

1. *POSTSCRIPT Language Reference Manual*, Adobe Systems Incorporated, Addison-Wesley, Reading, MA (1985).
2. "Adobe Type 1 Font Format," Part No. LPS0064, Adobe Systems, Inc. (March, 1990).
3. S. von Bechtolsheim, "The TEX PostScript Software Package," *TUGboat* **10**(1), p. 25–27 (1989).
4. N. Beebe, Private communication, via electronic mail. (1990).
5. L. Carr, "Of Metafont and PostScript," *TEXniques* **5**, p. 141–152 (August, 1987).
6. D. Henderson, "Outline fonts with METAFONT," *TUGboat* **10**(1), p. 36–38 (1989).
7. J.D. Hobby, "A METAFONT-like System with PostScript Output," *TUGboat* **10**(4), p. 505–512 (1989).
8. B.W. Kernighan, "A Typesetter-independent TROFF," Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, NJ 07974 (March, 1982).
9. D.E. Knuth, *The TEXbook*, Addison-Wesley, Reading, MA (1984).
10. D.E. Knuth, *The* METAFONT*book*, Addison-Wesley, Reading, MA (1986).
11. D.E. Knuth, METAFONT*: The Program*, Addison-Wesley, Reading, MA (1987).
12. G. Toal, Private communication, via electronic mail. (1990).
13. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* **3**, p. 337–343 (May, 1977).

◊ Shimon Yanai
IBM Science and Technology Center
Technion City

Haifa 32000
Israel
yanai@israearn.bitnet

◊ Daniel M. Berry
Computer Science
Technion
Haifa 32000
Israel
dberry@cs.technion.ac.il