Parametrization of PostScript fonts through METAFONT-an alternative to Adobe Multiple Master fonts

Yannis Haralambous

▶ To cite this version:

Yannis Haralambous. Parametrization of PostScript fonts through METAFONT-an alternative to Adobe Multiple Master fonts. Electronic Publishing, 1993, 6 (3), pp.145-157. hal-02100464

HAL Id: hal-02100464
https://hal.science/hal-02100464

Submitted on 25 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ELECTRONIC PUBLISHING, VOL. 6(3), 145–157 (SEPTEMBER 1993)

Parametrization of PostScript fonts through METAFONT — an alternative to Adobe Multiple Master fonts

YANNIS HARALAMBOUS

Institut National des Langues et Civilisations Orientales, Paris
Private address: 187, rue Nationale, 59800 Lille, France

email: Yannis.Haralambous@univ-lille1.fr

SUMMARY

In this paper we present a new method of parametrizing PostScript fonts in order to create font families. By changing parameter values one can obtain different weights, condensed or expanded versions, small caps as well as optically scaled fonts. The tool used to parametrize PostScript fonts is D. E. Knuth's METAFONT program. Instead of designing a font from scratch, METAFONT is used as an extrapolator of existing PostScript fonts: out of the information contained in them we build a meta-font; for every choice of parameter values, special versions of METAFONT allow us to return to PostScript and produce a new PostScript font.

KEY WORDS Font design PostScript METAFONT

1 INTRODUCTION

Browsing through the lovely book *Les caractères de l'Imprimerie nationale*, one remarks that the configurations of classical font families were quite different at the time of their creators than they are today (although names have not changed). For example, the *Garamond* face, made by Claude Garamond in 1530–1540, was available in roman/italics (6–36 points) and small capitals (6–20 points).

Bold and other weights were missing. This is not only a custom of the 16th–17th centuries. The *Marcellin-Legrand* face, made by Marcellin Legrand in 1825 and replacing the *Didot* face (1811) is available in roman/italics (4–28 points) and initials (10–64 points, for titling).

Again, just a single weight in many different point sizes. Nowadays PostScript font families are available in entirely different configurations. One has different weights (ranging from Extra-Light to Heavy or Ultra Black) as well as condensed versions, but small caps are underutilized: in rare cases there is at most one small capitals font. Furthermore there is no indication of point size: PostScript allows continuous linear scaling of fonts — this feature may be very useful for titling and graphic applications, but can hardly be used as a substitute for the many different point sizes needed for high quality typography.[1] The

[1] The different point sizes of traditional fonts had a different design, adapted to their 'real' size – PostScript fonts are scaleable and hence have no 'real size'; every point size is a scaled version of the same generic design.

0894–3982/93/030145–13$11.50
©1993 by John Wiley & Sons, Ltd.
Received 15 August 1993
Revised 1 December 1993

situation can be summarized as: *We use typefaces of older times, perverted so that they match the restrictions and pitfalls of modern technology*.

Several attempts have been made to counterbalance these restrictions (see [1], [2]) in the PostScript Type 1 font context. Furthermore, outside the strict context of Type 1 PostScript fonts there are two major solutions: the first, rather recent, is the new concept of *Multiple Master Fonts* introduced by Adobe in 1992; the second, almost 15 years old, is the font creation program METAFONT by D. E. Knuth.

This paper presents an intermediate solution. The traditional TEX & METAFONT process of font development and use is to build a font from scratch and to use METAFONT to create bitmap files (in a special format called PK) which are then used inside the TEX system. Instead of that, we propose the following procedure: (1) take an existing PostScript font, (2) convert it to METAFONT, (3) parametrize it as if it were a METAFONT font, (4) go back to PostScript, or even to Multiple Masters. In this way the power of METAFONT can be used efficiently in the PostScript font development context, complementing features which are not available in the existing formats.

1.1 Multiple Master fonts

Multiple Master Fonts were introduced in 1992, by Adobe Systems. They are an extension of PostScript Type 1 fonts (for a description of their format, see [3]). To create a Multiple Master font, one needs from 1 to 8 pairs of *master designs*. These are normal PostScript fonts, with very strong common properties (same number of control points for each character, etc.) and for a good reason: the Multiple Master font will be a *weighted linear interpolation* of the master designs.

This means that one can predefine up to 16 versions of every character, and for every choice of a quadruple of coefficients $(\alpha_1,\alpha_2,\alpha_3,\alpha_4) \in [0,1]^4$ one will obtain the corresponding intermediate character from a weighted linear interpolation. The restrictions on the master designs are intended to guarantee that every intermediate object is a valid PostScript Type 1 font.

The master designs can be either *extremal*, or *intermediate*. Usually the four degrees of freedom are used for *weight*, *width*, *optical size* and *style*. If one decides to use all four degrees of freedom then there are no master designs left for intermediate use, and only extremal ones can be used. There is an obvious risk in doing this: *can one be sure that the linear interpolation of an Extra-Light Garamond and an Ultra-Black Garamond* is *a decent regular Garamond font?* To avoid this risk one has to limit the range of variation to a secure minimum.

Perhaps — as we will see in this paper — the most important problem is that the different quantities involved in font design do *not* vary *linearly*. Measurements made on acceptable real-world font families show that a quadratic approximation gives significantly better results than a linear one.[2] An example of compared linear and quadratic approximations can be seen in Figure 5.

The last problem is that bold fonts often need a different number of control points than light ones; control points sometimes disappear in smaller point sizes where character shapes are supposed to get simpler. Interpolation then becomes impossible, since every control point of the final font is the interpolation of the corresponding control points of master

[2] This may have to do with the properties of human vision; the author would be glad to find out more information on this topic.

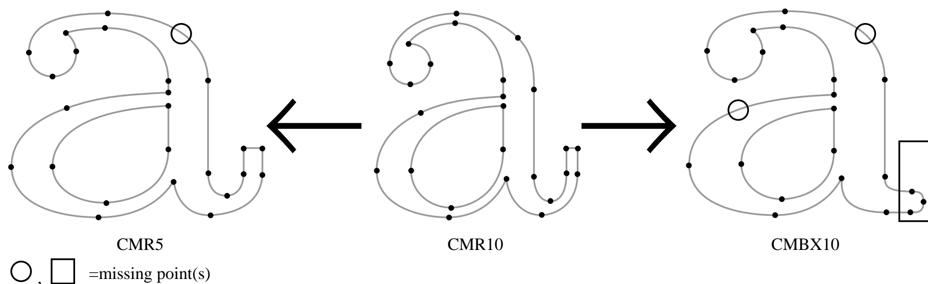

Figure 1. Candidates for master designs: Blue Sky Research `CMR5`*,* `CMR10`*,* `CMBX10`

designs. An example can be seen in Figure 1: the letters are taken from PostScript versions of fonts `CMR5`, `CMR10`, `CMBX10` of the Computer Modern Family (Blue Sky Research); these fonts could be ideal candidates for master designs: on the left the variation would correspond to the optical size axis (5 points → 10 points), and on the right to the weight axis (roman → bold extended). Nevertheless, as seen in the figure, the control points do not satisfy the required consistency conditions: the Multiple Master concept is not flexible enough to handle this case.

What are the advantages of Multiple Masters fonts? They are *interactive.* One can see intermediate versions by moving a cursor on the screen. In future software one will perhaps be able to see the variation inside a document — or software may be able to adjust the grey level automatically. But these are the standard WYSIWYG advantages; and the publishing world knows by now that WYSIWYG may be excellent for DTP and graphic arts, but is *not* always the best way to high-quality typography.

1.2 METAFONT

To match the quality of the TEX typesetting system developed in the late seventies by D. E. Knuth, the same author developed a companion program for font design and rasterization: METAFONT. As a first example, he produced an entirely parametrized font family, which is now distributed along with TEX.

The font family developed by Knuth is called *Computer Modern*. It is a derivative of the Monotype Modern typeface. Besides being a typeface used world-wide, Computer Modern is a magnificent *experiment.* METAFONT allows parametrization of every — even the tiniest — part of a character. One can define arbitrary metric relations between parts of a character, or between parts of different characters. By giving values to these parameters one can change a complete typeface in a homogeneous way. To demonstrate this enormous feature of METAFONT, Knuth produced all possible variations of the basic Computer Modern font, always from the same code, by changing the values of parameters. Besides weight variation and optical scaling, he also produced sans serif and typewriter fonts, as well as all kinds of exotic or funny fonts, out of the roman ones.

The infinite flexibility of Computer Modern fonts made their METAFONT code quite complicated and difficult for a novice to approach. One of the side-effects of this was of a psychological nature: nobody ever tried to restart such a project, and Computer Modern remained the only major Latin-script font family coded in METAFONT.

But METAFONT can be used in other ways. It is not necessary to start the design of a font from scratch, as Knuth did. Thanks to a set of utilities by Erik-Jan Vens, one can convert a PostScript Type 1 font into raw METAFONT code. This code can then be parametrized so that out of one or two different fonts one can interpolate (and extrapolate) a complete font family. The main variations that can be achieved concern scaling (expansion/condensing), weight (light/bold) and optical scaling (variable shape according to point size). Afterwards, the METAFONT code of fonts created in this way can be converted back to PostScript.

In this article we propose to describe the whole process in detail, by taking a simple but realistic example: a few letters out of the Bodoni font of the Serials collection, designed by Hans Florenz Walter Brendel (1991 B & P Graphics, Ltd.). Besides being beautifully designed, this family has the advantage of already proposing 7 differently weighted fonts, from 'light' to 'heavy'. In the following sections we will discuss in details the consecutive steps:

1. parametrization of the PostScript code,
2. rewriting the METAFONT code,
3. interpolation and extrapolation of weight,
4. condensing and extension,
5. small capitals,
6. optical scaling, and
7. going back to PostScript.

2 PARAMETRIZATION OF POSTSCRIPT CODE

PostScript Type 1 fonts use only two kinds of graphic element to describe characters: straight-line segments and third-degree Bézier curves. These combine to form boundaries of surface elements which are then filled with black. Such boundaries are called 'paths' and most of the commands available to describe a Type 1 character aim to abbreviate special kinds of these graphic elements. This code is converted by Erik-Jan Vens' utility `ps2mf` to very raw METAFONT code, looking more or less like this:

```
beginchar("i",295FX#,665FY#,0FY#); "i";
z1=(22FX,0FY);  z2=(278FX,0FY); % and other point definitions ...
fill z1 -- z2 -- z3 -- z4 -- z5 -- z6 -- z7 --
     z8 -- z9 -- z10 -- cycle; % filling a piecewise-linear
                              % closed path
fill z11 .. controls z12b and z13b .. z14
     .. controls z15b and z16b .. z17
                                & cycle; % filling a Bezier path
endchar;
```

The `beginchar` starts the description of character `i`, and assigns a width of 295 (horizontal) units, a height of 665 (vertical) units and zero depth to it. The instructions of the type $z_n = (x_n, y_n)$; define points `z1,...,z17` of the METAFONT coordinate system. In this way, points defined in the PostScript code have been given names z_n,[3] in their natural order. Their *x* and *y* coordinates are now multiplied by METAFONT quantities called `FX`

[3] And the PostScript *relative* coordinates have been replaced by *absolute* METAFONT coordinates.

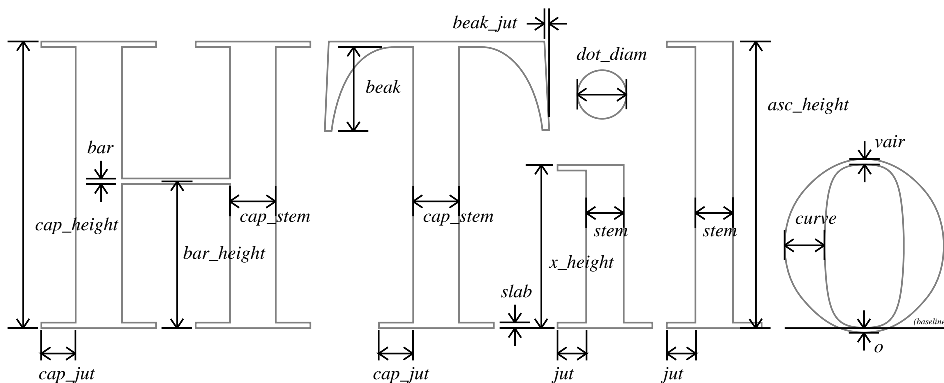

Figure 2. Names of character parts used in the modified METAFONT *code*

and FY. By modifying these, one can achieve instant scaling in the *x* or *y* direction — but we will see that there are better ways of doing this.

The `fill` commands fill the interiors of closed paths. The `--` operator produces a straight-line segment between two points, while the `..` operator produces a Bézier curve.

We know that a Bézier curve (being a third-degree polynomial) has four degrees of freedom. After specifying the beginning and end points, two degrees of freedom remain. Usually METAFONT takes its own initiative in the choice of the remaining data: one may ask for a curve leaving at a certain angle, or having a certain amount of tension, etc., and METAFONT will choose the 'nicest' Bézier curve out of all the curves satisfying the requested conditions. In our case, to be as close as possible to the original drawing, `ps2mf` has faithfully translated the information on the control points of the Bézier curve and is imposing these on METAFONT. This is the meaning of the `.. controls z1 and z2 ..` operator. When parametrizing our characters, we will bypass this operator, since an explicit determination of control points is often hard (and useless, except in rare exceptions).

Finally the `cycle` keyword means that the path should be closed. We will not enter more into the details of METAFONT syntax; the interested reader will find excellent presentations of this in [4] and [5].

The purpose of this section was to show to the reader how much information is involved in the description of each character, and how this information is converted from PostScript into raw METAFONT code. In the next sections we will see how to interpret, process and extrapolate this information to parametrize the characters.

3 REWRITING THE METAFONT CODE

The most important and most fascinating part of the work involved in parametrizing a font is to analyse the character shapes and find out which dimensions are significant for more than one character, if not for the whole font. To take some trivial examples: the width of the dot of i must be the same as that of the dot of j; the width of the vertical stem of I must be the same in the letters B, D, E, F, H, T . . . , but most certainly not the same as that of the i, and so on. Centuries ago, typeface creators had already classified many of these quantities and given them names. In Figure 2 we give the names we will use in our

examples of Bodoni characters. These are not necessarily traditional names, but are used by D. E. Knuth in [6] where he parametrizes the Computer Modern font family.

Some parameters may take the same value in the original PostScript font (for example *hair* and *vair*, or *curve* and *stem*). These will be differentiated when we start to produce variations of the font (in METAFONT jargon this operation is called 'adding metaness'). Generally it is the responsibility of the reader to decide if slightly different quantities should be identified; elementary aesthetic arguments — mostly involving symmetry — are often sufficient to reach a conclusion. For example we have chosen to identify the stroke width of all serifs and named this quantity *slab*, but we could also distinguish between uppercase and lowercase serifs, and so on. This shows that there can be an arbitrary number of parametrizations of the same PostScript font; it is up to the reader to find the correct balance between too many parameters (loss of homogeneity, unnecessary complication) and too few parameters (simplistic result, failure when extrapolating). The final goal is to *express the generic intentions of the original font designer* in METAFONT language.[4]

Once we have given names to character parts we have to study the (raw) METAFONT code to see if the dimensions of these parts are fixed throughout the font (since fonts are often created 'manually', that is by use of a tablet, these dimensions may vary slightly — this is also a means to evaluate the quality of a font). Once this is done, we start writing our new METAFONT code by defining these quantities and giving them initial values:

```
cap_height#:= 720FY#; asc_height#:= 720FY#; x_height#:= 410FY#;
desc_depth#:= 0;      bar_height#:= 0.5131cap_height#;
cap_stem#:=   94FY#;  stem#:=       72FX#;  curve#:=    81FX#;
slab#:=       14FY#;  vair#:=       14FY#;  bar#:=      slab#;
dot_diam#:=   108FX#; cap_jut#:=    86FY#;  jut#:=      72FX#;
beak#:=       209FY#; beak_jut#:=   10FX#;  o#:=        14FX#;
```

The # operator indicates that the quantity preceding it is a 'real' dimension (*sharp* dimension in METAFONT jargon). The next step will be to use the `define_pixels` command, to define pixel-dependent quantities, identified by the same names, without the hash mark.

Now we will modify the description of every character so that all the parameters we have defined appear, and not a single character part remains unparametrized. For the specific needs of our method we are bound to use only *filling* METAFONT commands, and not *drawing* ones. Nevertheless, the code will be quite simple and readable, since we are going to use a special category of objects, called 'simulated pens'.

The data involved in such an object is (1) a point of the plane (the centre of the pen), (2) a length dimension (the width of the pen) and (3) an angle (the angle of the pen). Intuitively, defining a simulated pen is the same as choosing a razor-like pen with the given width, turning it to the appropriate angle and positioning it onto the chosen point. Once all the simulated pens have been defined, one can ask METAFONT to draw a stroke passing through those pen positions. Behind the scenes, METAFONT will replace the command for 'drawing that stroke' by the usual `fill` command of the closed path defined by the pen

[4] Some readers may argue that font design is a purely artistic activity which cannot be modelled by mathematical equations. This may be the case for the first design of a font; but when the designer tries to extrapolate his/her font to produce faces of different weight or size, then he/she is certainly using mathematical rules to ensure homogeneity and uniformity. This is the part of artistic creation which our method aims to model, and no more; it is far less than the original goal of D. E. Knuth when creating METAFONT.

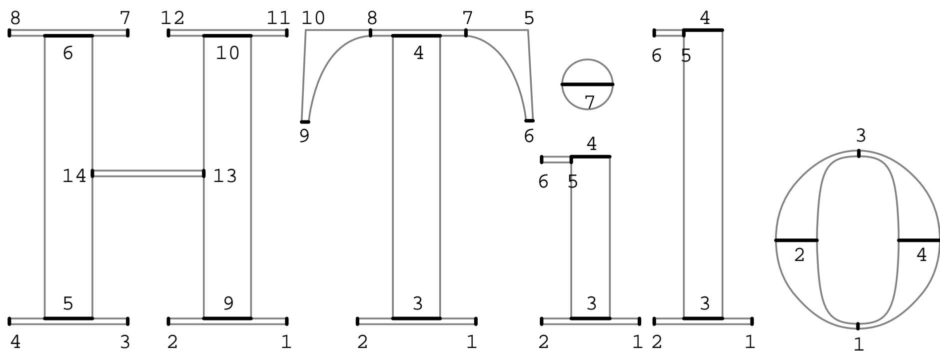

Figure 3. Loci of simulated pen positions

positions and the central path of the requested stroke. Since this operation involves only *filling*, we can use it without complications when going back to PostScript.

The next step will be to choose the locations of the (simulated) pen positions we are going to define. These can be seen in Figure 3. First of all we describe the pen positions without giving the exact coordinates of their locations. Here is the description of the pen positions for the letter i:

```
pos1(slab,90); pos2(slab,90); pos3(stem,0); pos4(stem,0);
pos5(slab,90); pos6(slab,90); pos7(dot_diam,0);
```

The syntax used is: `pos`$_{\mathtt{n}}$(⟨*width*⟩,⟨*angle*⟩); where `n` is the suffix of the centre of the (simulated) pen z_n. So for example pen positions 1 and 2 determine the serif of the letter; pen positions 3 and 4 determine its vertical stem, and so on.

Next we have to position the pens we have defined. Here, one of METAFONT's most convenient features appears. Instead of giving the coordinates of each point explicitly, it is sufficient to provide a system of linear equations between them. This spares the designer a lot of unnecessary calculations and allows a very flexible control of the character shape under the effect of varying parameters. We will describe the equations needed for the letter i in detail, since these provide a good introduction to the operations we are going to make in the rest of this article.

First of all, some syntax elements: suppose we have defined a pen by `pos3(stem,0);`. Then automatically `z3` is the centre of the pen, and `z3r,z3l` are the right and left edges of the pen. Furthermore, whenever `z3` is a point in the METAFONT plane, `x3` and `y3` are its *x* and *y* coordinates. Once the width and angle of the pen are known, the position of the pen can be determined by the coordinates of `z3` *or* `z3l` *or* `z3r`, or by any combination of one *x* and one *y* coordinate of these three.

Let's start with simulated pens 1 and 2 (of letter i). These are taken with an angle of 90 degrees. Their left edges are to be taken at the baseline: `y1l=y2l=0;`. The *x*-coordinates of these simulated pens are at the two sides of the character box: `x2=0; x1=w;`, where `w` is the width of the character box as defined in the `beginchar` command (`h`, `d` being the height and depth of the box).[5]

[5] The PostScript-accustomed reader should note that these are dimensions of an imaginary box which will be used by TEX to typeset; they do *not* correspond to the PostScript character bounding box whose dimensions appear in the `AFM` file.

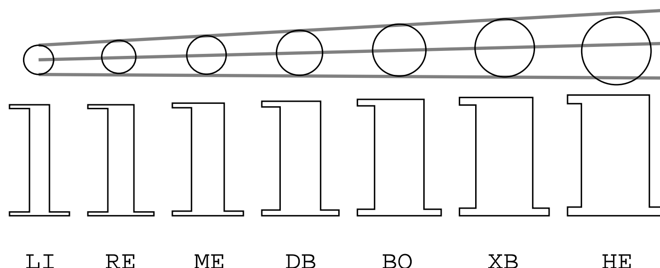

Figure 4. The letter i *in the seven weights of Brendel Bodoni*

Pens 3 and 4 will produce the central stem. They should be centred, and the same applies to pen 7: `x3=x4=x7=0.5w;`. Concerning the y-coordinate, pen 3 must be 'lying upon' the stroke joining pens 1 and 2, in other words `y3=y1r;`. Pen 4 must be at height `x_height`, since the dotless i must have the standard height of lowercase letters without ascenders: `y4=x_height;`. The right edge of pen 5 is identical to the left edge of pen 4: `z4l=z5r;`. The right edge of pen 6 is horizontally aligned to the right edge of pen 5: `y6r=y5r;` and the x-coordinate of pen 6 is equal to that of pen 2, according to the design of the Bodoni i: `x6=x2;`.[6]

The remaining dimension (`y7`, i.e. the height of the dot) is the only one needing special care. Having chosen the Brendel Bodoni font set, we already have 7 weight variations available (Light, Regular, Medium, Demi-Bold, Bold, Extra-Bold, Heavy). In Figure 4, the reader can compare the different i's, at *equal distance*. One sees immediately that the heights of dots (as well as the dot diameters) for the first six weights follow a linear rule. To help the reader in visualizing this property, we have drawn grey lines joining the dot centres and vertical dot extremities of weights Light and Extra-Bold; clearly the four intermediate ones fit between the lines, while the seventh (Heavy) disobeys this rule.

This intuitive result is verified by a linear approximation of the function `dot_height` $=$ $f($`dot_diam`$)$ for the first six weights. The values of dot centre heights and dot diameters as taken from the PostScript code are

weight	LI	RE	ME	DB	BO	XB	HE
dot_height	576	586	594	602	612	620	609
dot_diam	108	123	142	166	192	214	250

A fairly succesful approximation of the `dot_height` versus `dot_diameter` relationship is `dot_height` $\approx 535.5 + 0.4$ `dot_diameter`, with the exception of the seventh (Heavy) weight.

The linear equation above has a non-zero scalar part (535.5). This number is independent of the weight, but depends however on the character's size. Therefore we will express it by some vertical dimension of the character, with similar behaviour: the best choice seems to be `asc_height`. So finally we get a rule for the height of the dot of i:

`y7` $=$ `0.74375 asc_height` $+$ `0.4 dot_diam;`

where 0.74375 is the quotient 535.5/720 (720 being the ascender height for all weights).

After having completed the set of linear equations, we draw the characters. This is straightforward; one should only keep in mind that only `fill` operations will be used, so that post-conversion to PostScript is possible.[7]

[6] We know that the x-coordinate of pen 2 is equal to 0, but this may change – the fact that `x6=x2` may also change . . . as a matter of fact we must find the formula which is least likely to change when varying the letter.

[7] For reasons of space, we do not include the complete METAFONT code for letters H, T, i, l, o. This code, as

Figure 5. Quadratic (black) and linear (grey) interpolations of `stem` *and* `cap_stem` *parameters*

4 INTERPOLATION AND EXTRAPOLATION OF WEIGHT

The fact of having seven different weights of the same font is rather exceptional and one could argue that they cover all possible cases where this typeface can be used. This is actually wrong, since often one needs a font with a specific grey density level. In that case it would be nice to be able to vary the weight of the font continuously until the right grey density is obtained.

Once again we will examine the values of parameters for the seven available weights and try to interpolate and extrapolate them. In Figure 5 the reader can compare the results of a quadratic interpolation (in black) and of a linear one (in grey) on parameters `stem` and `cap_stem`. We consider this interpolation sufficiently succesful to base both parameters on that rule. As a matter of fact we will make out of `stem` and `cap_stem` two *generic* parameters; this means that we are going to interpolate other parameters of each letter with respect to `stem` if the letter is lowercase or with respect to `cap_stem` if the letter is uppercase.

We have applied this method to parameters `slab`, `jut`, `dot_diam`, `curve`, `vair`, as well as to the widths of H, T and o. The quadratic approximations we obtained have been used as general interpolation rules; the reader can see the result in the following 51 words, which represent fonts going from an Extra-Light weight to the original Heavy one.

HTilo, **HTilo, HTilo,**

5 CONDENSING AND EXTENSION

Many desktop publishing or word-processing programs allow the user to condense or extend PostScript Type 1 fonts; in reality this operation is a horizontal scaling. This means

well as all METAFONT files used for the examples in this paper, can be obtained by anonymous ftp from `ftp.ens.fr` (IP 129.199.104.12), directory `/pub/tex/yannis/ridt94`.

that not only the width of characters but also the width of stems and curves, and every horizontal dimension of the characters, is scaled as well. Another possible way of scaling is to leave weight (widths of stems) unchanged: one should be able to change the width of a character while keeping the widths of the principal strokes fixed.

This is trivial for METAFONT, since every width is separately parametrized. To scale characters horizontally we just need to change their global widths; this does not affect any other parameter. However, for aesthetic reasons we have chosen to modify the width of serifs as well (parameters `jut` and `cap_jut`) but to a lesser extent than the character's global width.

In the next examples, the reader can compare 'PostScript-like' global condensing and extension (horizontal scaling of all character properties) with 'METAFONT-like' selective horizontal scaling:

	Selective scaling					*Global scaling*				
	81%	90%	100%	111%	123%	81%	90%	100%	111%	123%
LI	HTilo	HTilo	HTilo	HTilo	HTilo	HTilo	HTilo	HTilo	HTilo	HTilo
ME	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**
BO	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**	**HTilo**

6 SMALL CAPITALS

Small capital letters ('small caps') are another weak point of the PostScript font scene. Obviously, it is bad typographical practice to use uppercase characters from a linearly reduced font as small caps; nevertheless this is what happens most of the time, for very obvious reasons: firstly, there are not many ready-to-use small caps fonts, and secondly modifying reduced uppercase letters to look like small caps is not a trivial issue.

In our context, making small caps is straightforward. One only needs to replace uppercase parameters by the corresponding lowercase ones (more-or-less, since the height of small caps is usually a little more than the height of lowercase characters). In particular, one obtains small caps for *all* weights, a feature that — at least to the author's knowledge — is offered by no current PostScript font family.

Here is how to proceed: the parameters involved in uppercase character construction are `cap_height`, `bar_height`, `cap_stem`, `beak`, `beak_jut` and `cap_jut` (see Figure 2), as well as the character widths. We will make the following replacements:

- `cap_stem` and `cap_jut` will be replaced by `stem` and `jut`;
- in theory, we could replace `cap_height` by `x_height` and scale the other parameters accordingly. However, small caps are usually slightly higher than lowercase characters; D. E. Knuth scales this height by a factor of 1.1935 (cf. [6, p. 30]) and we will – once more – follow his example;
- the parameter `slab` will *not* be changed. As a matter of fact, the author has all too often heard traditional typographers say: 'There will never be a computer Bodoni [*sic*] since after photomechanical reduction thin strokes (*traits déliés*) disappear'. This is exactly what we avoid by keeping the value of `slab` constant.

The caps and small caps versions of letters H and T of our example follow, in 14 different weights:

7 OPTICAL SCALING

Perhaps the most important application of font parametrization is optical scaling. In older times, different sizes of typefaces were cut separately in order to optimize their readability and improve the overall appearance of printed pages with more than one point sizes. PostScript fonts can only be linearly scaled: smaller sizes appear narrower than normal (for example in the case of footnotes)[8] and are hard to read; bigger sizes appear too large and their thin strokes too thick.

When a font is *optically scaled*, the shape of characters varies as in traditional font design. Usually fonts are designed at 10 points. When the font is scaled to bigger point sizes, widths of strokes as well as the character's global width are scaled to a lesser extent than the character's height, making it look lighter — titles look natural and do not hurt the eye. On the other hand, when the font is reduced (for example for footnotes or marginal notes) the thin strokes are reduced less than the thick strokes and tend to converge to a certain limit so that they always remain visible; also letters are wider, interletter spacing is increased and ligatures like fi, fl are broken, to increase readability.

To achieve optical scaling we will apply general rules (again quadratic approximations to known examples) to create distinct fonts for every point size. The task of making the necessary measurements has been done by D. E. Knuth on Monotype Modern 8A (cf. [6, p. vi]). We will take his results and apply them to our case, since optical scaling coefficients have more to do with the properties of human vision than with the style of the font.[9]

There follows an example of D. E. Knuth's coefficients for the Computer Modern Roman font family, the numbers given being factors applied to the corresponding quantity *after* a linear scale:

point size	17.28	12	**10**	9	8	7	6	5
`cap_height`	1	1	**1**	1	1	1	1	1
`cap_stem`	0.723	0.912	**1**	1.01	1.03	1.07	1.12	1.19
`slab`	0.842	0.947	**1**	1.02	1.08	1.17	1.29	1.45

These numbers are quite significant: both `cap_stem` and `slab` are reduced for bigger point sizes and extended for smaller point sizes, but following different numerical rules. To achieve a satisfactory quadratic approximation to the optical correction factor applied to `cap_stem` and `slab`, with respect to point size, we have treated separately point sizes bigger and smaller than 10 points. The reader can see the results in Figure 6.

All parameters of our five-letter example appear in the Computer Modern fonts; for each of them we have made similar quadratic approximations of the optical correction factor.

[8] As this one, which has been typeset in the currently most standard PostScript font: Adobe Times Roman.

[9] The author would be very grateful for the results of similar measurements on other traditional fonts, to either confirm the method of D. E. Knuth, or propose alternative numerical rules.

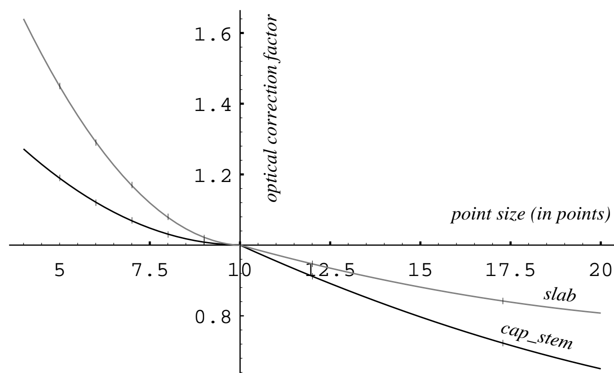

Figure 6. Piecewise quadratic approximations of optical correction factor/point size

The reader can compare the results in the next table, where in the first column appear the non-optically corrected linearly scaled characters, in the middle column the optically corrected ones, and in the right column the optically corrected ones, magnified so that they have a real size of 17.28 points.

Non optically scaled	Optically scaled	
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo **HTilo**	HTilo **HTilo**
HTilo **HTilo**	HTilo HTilo	HTilo HTilo

8 GOING BACK TO POSTSCRIPT

After having parametrized the METAFONT code sufficiently to obtain all desirable variations, we return to PostScript by using special versions of METAFONT with PostScript output. The important fact in this conversion is that all the METAFONT used in our code has its PostScript Type 1 equivalents: once the parameters have received their values and all calculations have been done, the operations METAFONT has to make are `fills` of areas delimited by Bézier curves — this is exactly what PostScript Type 1 code can do (and no more).

Among special versions of METAFONT with PostScript code we can list the following:[10]

[10] The author is trying to convince Wilfried Ricken (`wilfr@hadron.tp2.ruhr-uni-bochum.de`) to include `mf2ps` support in DirectTEX; hopefully by the time of the RIDT94 conference he will have suceeded.

- MacMETAFONT by Victor Ostromoukhov; for the Macintosh, available as an MPW tool. Freeware.
- mf2ps, by Shimon Yanai and Daniel Berry (cf. [7], [8]); available as changefile for the METAFONT Pascal-WEB source code. Freeware.
- MetaPost, by John Hobby (cf. [9]); extension of METAFONT with PostScript output only. Written in C-WEB AT&T.

All of them produce PostScript Type 3 code. Many commercial utilities allow conversion into Type 1 code and (poor) automatic hinting. A very interesting further development of this method would be hint generation from METAFONT raster optimization techniques and parametrization: in this paper these techniques have not been examined since the final goal was to obtain Type 1 code: abstract Bézier curves, and not bitmaps — which are METAFONT's standard output.

REFERENCES

1. Jacques André and Corinna Kinchin, 'Adapting character shape to point size', *PostScript Review*, 31–36, (1991).
2. Jacques André and Irène Vatton, 'Contextual typesetting of mathematical symbols taking care of optical scaling', INRIA/IRISA-Rennes, Research report no. 1972, (1993). (Submitted to *Electronic Publishing*.)
3. Adobe Developer Support, *Adobe Type 1 Font Format: Multiple Master Extensions*, 1992.
4. D. E. Knuth, *The* METAFONT*book* (Volume C of *Computers and Typesetting*), Addison-Wesley, Reading, MA, 1986.
5. Helmut Kopka, LaTeX*, Erweiterungsmöglichkeiten*, Addison-Wesley, München, 1991. Second edition.
6. D. E. Knuth, *Computer Modern Typefaces* (Volume E of *Computers and Typesetting*), Addison-Wesley, Reading, MA, 1986.
7. Shimon Yanai, *Environment for Translating* METAFONT *to PostScript*, Ph.D. dissertation, Israel Institute of Technology, Haifa, 1989. In Hebrew.
8. Daniel Berry and Shimon Yanai, 'Environment for translating METAFONT to PostScript', *TUGboat*, **11**(4), 525–542, (1990).
9. John Hobby, 'Introduction to MetaPost', in *Proceedings of the 7th European TeX Conference*, pp. 21–36, Prague, (1992).