ELECTRONIC PUBLISHING, VOL. 7(2), 1–999 (JULY 1994)

Dynamic optical scaling and variable sized characters

JACQUES ANDRÉ
IRISA/INRIA–RENNES
Campus Universitaire de Beaulieu
F–35042 Rennes cedex, France
email: jacques.andre@irisa.fr

IRÈNE VATTON
CNRS/INRIA–RHÔNES-ALPES
2 rue de Vignate
F–38610 Gières, France
email: irene.vatton@imag.fr

SUMMARY

First, a survey on optical scaling is carried out, both from the traditional point of view and from that of today's digital typography. Then the special case of large characters, such as braces or integral signs, is considered. It is shown that such variable sized symbols should be computed at print time in order to approach the quality of metal typesetting. Finally, an implementation of such dynamic fonts, still in progress in the Grif editor, is described.

KEY WORDS Optical scaling, variable sized characters, large symbols, dynamic font, parametrized font, Grif

1 INTRODUCTION

The expression "optical scaling" refers to a concept known by type designers for centuries, even if this expression is relatively new and not very suitable[1]: metal characters differ from one body to another one not only in size, but in shape too. Figure 1 shows various Garamond "e" at different point size but photographically magnified to the same size. It is obvious that the "e" is, relatively, fatter at 8 point size than at 36 point size. On the other hand, the counter (the white inner part) is, relatively, larger at 8 point size than at 36 point size. The main reasons are : with small characters, thinner strokes would be unreadable, and smaller counters would result in inkspreading (the white part would be filled in with ink and would get black); on the other hand, if the strokes of large characters were as fat as small ones, characters would look bolder. This "optical scaling" applies not only to character shapes, but to their metrics: the smaller a character, the bigger its relative width.

With metal types, this was not difficult to achieve: each punch was cut separately. Richard Southall says [2]: "In hand punchcutting, the character shape on the face of the punch is made in its final size. An important consequence of working at final size, rather than making large drawings and reducing them as happened in later techniques, is that the punchcutter does not have to take into account the effect that reduction in size has on the appearance of a shape."

[1] This expression enables one to believe in the "zooming" process, i.e. in linear scaling which is the opposite of optical scaling. Futhermore, the confusion with "optical correction", sometimes called "optical compensation" as well, should be avoided. These last two expressions mean that character shapes are drawn in such a way as to give the reader the perception that he sees what his mind expects to see (e.g. a letter O positionned on the base line) rather than what the character looks like (e.g. the baseline generally overhangs the bottom of a letter O). See [1] for more examples.

0894–3982/94/020001–999$504.50

Received June 1993
Revised June 1994

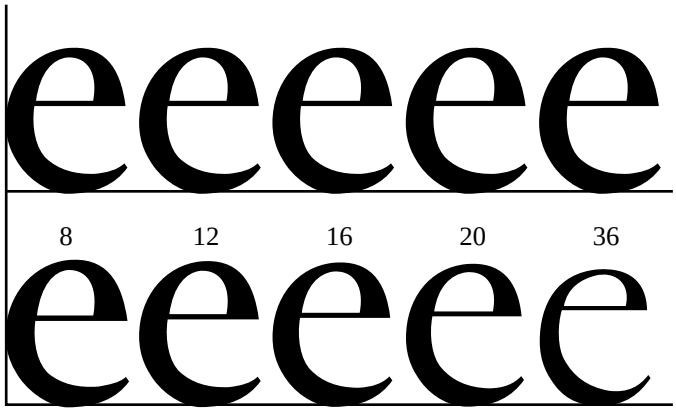

Figure 1. Top: Adobe Garamond characters, from 8 to 36 point size, magnified at the same size; bottom: genuine Garamond characters, photographied, digitalized then magnified to the same size: note the different shapes produced by optical scaling.

This quality was lost when punchcutting became an automatic process thanks to the pantographic punchcutting machine designed by Lynn Boyd Benton in 1885 (see [2]): only one shape was used for a given character at any point size. Today, the process is the same even if techniques have changed: it consists in describing the contour of a large character, generally at 1000 point size, then to use a scaling factor to reduce it to the requested size (see [3], [4] and [5]). Obviously, this results in the same shape. Figure 1 (top) shows the same Adobe-Garamond[2] "e" at different point sizes but magnified to the same size: all these "e" are identical.

Since 1991, various commercial products, such as *Multiple Master Fonts* by Adobe [7,8], or *MultiType* [9,10], etc. have been proposed in order to retrieve the metal type quality by allowing optical scaling. Note that optical scaling has been offered since 1978 by METAFONT, Knuth's system to design fonts [11].

After describing in more detail the way optical scaling is performed by a computer, another way of adjusting character shape according to the point size will be proposed. Then this paper shows how this technique can be applied to a special class of characters: the large symbols such as braces, parentheses and mathematical symbols.

2 OPTICAL SCALING AND DIGITAL FONTS

2.1 Linear scaling

First, let us recall how character contours are described and rendered [4].

A character (like the "e" in figure 2-left) is composed og a set of contours that are filled (here with grey ink). These contours are paths defined as sets of segments and of splines, generally Bézier curves. For example, the inner counter is defined as follows:

1. start from point number 10
2. proceed to point number 11 following the Bézier curve defined by the two extremity

[2] This font has been compared to the genuine Garamond by Mark Agetsinger[6].

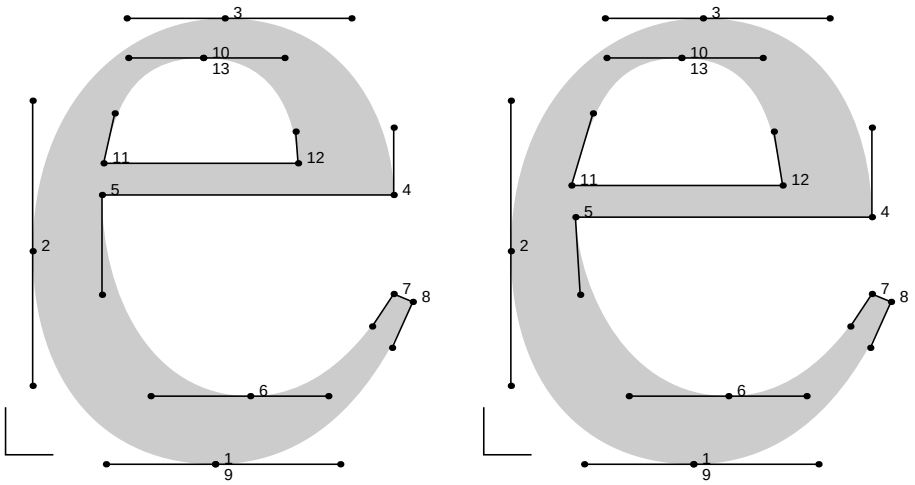

Figure 2. Times "e": left, the regular description using Bézier curves; right: the modification of coordinates of points number 4, 5, 11 and 13 results in an enlarged counter

points 10 and 11, and by the two control points (bullets at the ends of the tangents to the curve)

3. draw a straight line from point number 11 to point number 12
4. proceed to point number 13 following the Bézier curve defined by points number 12 and number 13 and the two corresponding control points.

Generally the coordinates of these points are given in a coordinate system where characters have 1000 point size. In a PostScript like language, the "e" can be programmed as follows:

```
Bezier curve from x=248 y=501                % point 10
to x=123 y=368                               % point 11
with control points  x=154 y=501 x=137 y=431

line from x=123 y=368                        % point 11
to x=367 y=368                               % point 12

Bezier curve from  x=367 y=368               % point 12
to  x=248 y=501                              % point 13
with  control points x=364 y=408 x=350 y=501
```

This resolution-independent definition is scaled according to the specific requested size. The rendering algorithms include outline scan conversion and filling which are called and generate a bitmap that is copied at the requested position of the page image.

Because coordinates such as $x = 248, y = 501$ have numerical values which are independant of the actual body size, it is obvious that the shapes produced at various sizes are identical (like all the Adobe-Garamond "e" of figure 1-top).

2.2 Analytical description

Languages like PostScript or METAFONT do not only support numerical values for defining outline control points. For example, it is possible to replace the numerical coordinates of points 4, 5, 11 and 13 by expressions or functions whose value depends on the actual body size. The above counter contour may then be written as follows:

```
Bezier curve from x=248 y=501                          % point 10
to x=f11x(bodysize) y=f11y(bodysize)                   % point 11
with control points  x=154 y=501 x=137 y=431

line from  x=f11x(bodysize) y=f11y(bodysize)           % point 11
to  x=f12x(bodysize) y=f12y(bodysize)                  % point 12

Bezier curve from  x=f12x(bodysize) y=f12y(bodysize)   % point 12
to  x=248 y=501                                        % point 13
with  control points x=364 y=408 x=350 y=501
```

Running this program twice with two different values of `bodysize` gives the two different shapes of figure 2.

This shows the way all optical scaling systems work. Fonts enabling such size-dependent variations are sometimes called “parametrized fonts”.

2.3 Which set of points?

Let us assume that the body size is the only factor that affects the shape of a character[3]. Any point entering the definition of an outline has coordinates x and y such that $x = f(bodysize), y = f'(bodysize)$. The main problem of optical scaling is then to determine these functions f, f'.

Very few studies have been carried out and published about such functions. We may only quote a thesis written by Bridget Lynn Johnson [12] and some measurements by Donald Knuth ([13, page vi] quoted and analyzed by Haralambous [14]) on Monotype Modern. Their results are confirmed by what seems to have been implemented in the commercial products quoted above, even if these results are criticizable[4]. On the other hand, while the properties of human vision are highly important, it does not seem that definitive results can be obtained from research in human vision and perception.

2.4 Expansion factor

We call expansion factor of a given measure (e.g. the height of a given character, or its width, or the abscissa of a given point), at a given size, the ratio of this measure when optical scaling is applied over the same measure without optical scaling. In figure 2 for example,

[3] Optical scaling is concerned not only with metric relations between parts of a given character but also with the metrics of the character itself (e.g. its x-width) and even with metrics between parts of subsequent characters.

[4] For example, Johnson measured printed material instead of smoke proofs which would have given better results. Futhermore, some authors, like Southall [2], think that the contour of a character is not the appropriate concept to be used.

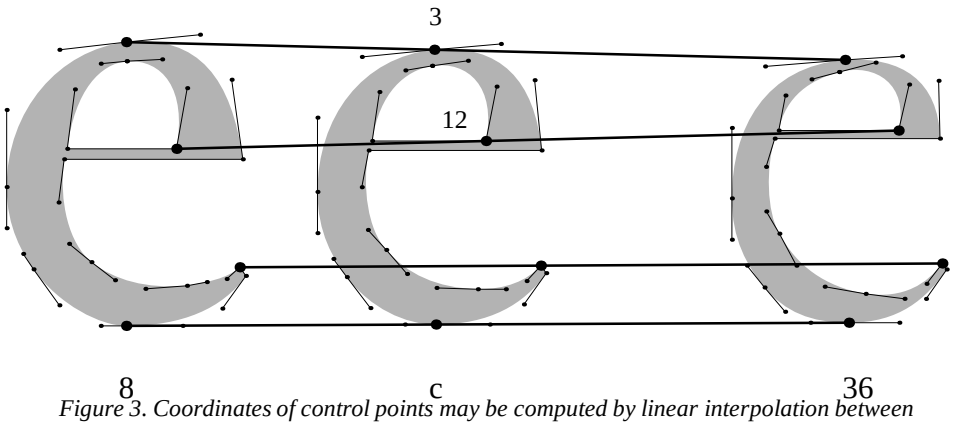

Figure 3. Coordinates of control points may be computed by linear interpolation between corresponding points – see figure 1

assuming that the right "e" is at 8 point size ($bodysize = 8$) and that $f12y(8) = 340$, we get for this "e", the two following expansion factors F:

$$F_{\text{ordinate of point 12}} = \frac{340\text{pt}}{368\text{pt}} = 0.924$$

$$F_{\text{counter height } (y_{10}-y_{12})} = \frac{501-340}{501-368} = 1.211$$

This last factor for the genuine Garamond compared to Adobe-Garamond (figure 1) gives:

$$F_{\text{Garamond counter height at 8 pt}} = \frac{0.6\text{cm}}{0.57\text{cm}} = 1.053$$

$$F_{\text{Garamond counter height at 36 pt}} = \frac{0.51\text{cm}}{0.57\text{cm}} = 0.896$$

2.4.1 Linear interpolation of control points at regular sizes

In the usual range of body sizes (say from 6 points to 36 points), it seems that a good approximation is to say that expansion factors F are linear functions of the body size:

$$F_{bodysize} = \alpha \times bodysize + \beta$$

Notably, the positions of control points follow a linear rule. For example, in figure 3, having the coordinates of point number 12 at 8 point size and at 36 point size, the coordinates of this same point number 12 at c point size can be obtained by linear interpolation.

It is important to quote that each linear rule is specific to a given measure. That is why (figure 3) the set of points for the right end of the counter (point number 12) is not parallel to the set of points of the upper point of this counter (point number 3).

2.4.2 Small characters need quadratic functions

Both D. Knuth and B. Johnson agree that expansion factors are no longer a linear function of the body size when this size is small. Quadratic interpolation is needed. Knuth uses 10 point size as the boundary: expansion factors are far larger when the body size becomes smaller (see figure in [14, page 156]).

2.4.3 Non identical sets of points

Actually, the main problem is that contours are described with splines, i.e. with sequences of curves. This set of curves may differ from one character at a given point size to the same character at another point size (typical examples are given in [14, figure 1]). Small characters cannot support all topological attributes of large characters (e.g. serifs are simpler); furthermore, at a larger point size, one character section given by a single large Bézier curve is better represented by two smaller Bézier curves.

2.5 Large symbols

Optical scaling studies and implementations deal with letters whose expansion factors are roughly in the range $[0.5, 2]$. For example, the "e" Garamond width expansion factor at 100 point size is less than 2 times smaller than the same factor at 5 point size. This is not true for “large symbols” where the expansion factors may change in a ratio larger than 20.

A typical example concerns braces. Figure 4 shows the same brace from 1 to 27 *douzes* which is the old French name for the multiple of twelve didot points (the real set goes up to 50 *douzes*). All of these braces have swellings with the same thickness. Here, this thickness is called *corps* (bodysize) and has a value of 6 points (i.e. six-to-em). Other plates exhibit other sets of braces, with a thickness from 2 to 10 points (3 point and 6 point braces are most commonly available in such catalogues). Finally, each brace may be characterized by two parameters, its height h and its thickness t, with $h \in [1:50], t \in [2:10]$.

For a given thickness, say $t = 6$, there are 50 different heights. It can be said that the expansion factor goes from 1 to 50, i.e. with a ratio equal to 50, which is far bigger than for letters.

Font founders offered other such symbols, e.g. parentheses, rules, angular brackets, etc. Another class of large symbols concerns mathematical variable-sized symbols such as integral sign, summation sign, brackets, arrows, radical sign, etc. These symbols have to follow optical scaling rules: if a 50 point size integral sign was magnified through some affine transformation to 100 point size, this magnification would apply as well to its thickness: the stem would be excessively bold (figure 5).

2.5.1 Computerized mathematical fonts

Although there are thousands of fonts available on laser printers or phototypesetters, only few of them are concerned with mathematics or with large symbols. Today, the main ones are (see [15, page340]):

- *cmex*nn (math extension) fonts created by METAFONT for TEX ([16, appendix F]);
- *Symbol* font, one of the four fonts that have always been installated on any PostScript system ; refer to [17, appendix E.11];

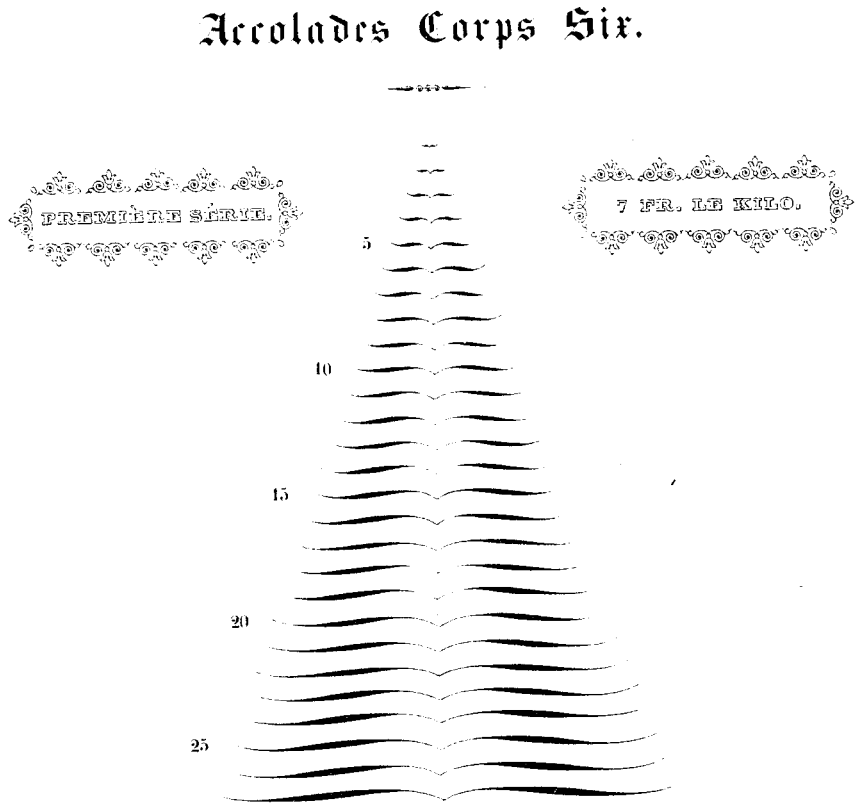

Figure 4. Metal founders offered large sets of variable-sized symbols, at various thicknesses, and for each, at different sizes. Here, a plate of six point braces from the Fonderie Générale*, around 1935 (courtesy* Musée de l'imprimerie, Lyon*).*

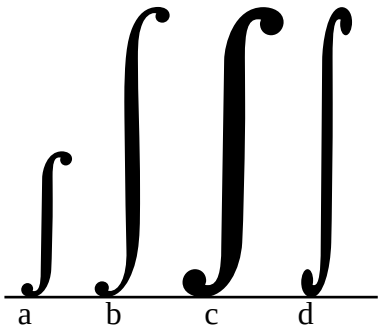

Figure 5. a) "text integral" at 50 point size
b) correct "display integral" with the same scaling factor (50 point size)
c) wrong display integral taken from a text integral magnified 2 times (stem too fat)
d) wrong display integral taken from a text integral scaled by a factor of 2 in y only (deformed spurs)

- The *Lucida* family that offers both a *LucidaMath-Symbol* almost equivalent to Adobe's *Symbol* and a *LucidaMath-Extension* where one can find symbols that belong to TEX's *cmex* [18].
- Two new versions of *Times* (by Spivak and by Jungers) that offer TEX's math extension as *Lucida* does.

2.5.2 Remaining problems

Today, rather good quality formulae can be drawn with text formatters. However some problems remain.

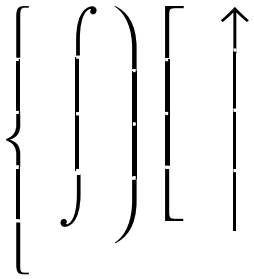

*Figure 6. Variable-sized symbols are generally drawn as composite symbols. The spaces between items are only present here to indicate the miscellaneous parts of symbols. (*Symbol *font).*

```
\overbrace{x} = \overbrace{a+b+c+d}  ⇒  $\overbrace{x} = \overbrace{a+b+c+d}$
    \widehat{x} = \widehat{a+b+c+d}  ⇒  $\widehat{x} = a+\widehat{b+c}+d$
```

Figure 7. LaTeX *enables some horizontal symbols such as "over braces" to be extended, but produces unexpected results with non extensible symbols such as "wide hat".*

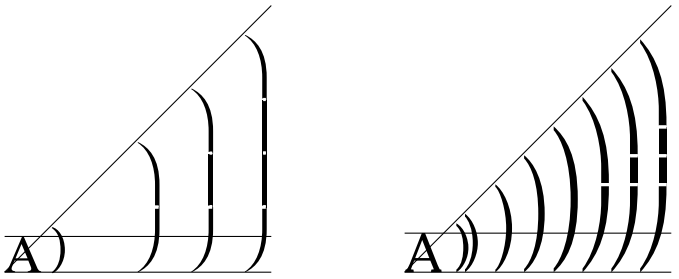

Figure 8. Gaps in variable-sized symbols ar more important in Symbol *(left) than in* Lucida *(right). The body size is indicated by the "A" cap height.*

- While mathematics oriented hot metal fonts contained a large set of symbols at any point size, computerized fonts replace variable-sized symbols by a discrete set of composite symbols. For example, the ends of an integral sign are made of the upper spur and of the lower spur while the stem is assembled with a set of spare bars (figure 6).
- Good mathematics require slanted integral signs. *Symbol* does not offer such slanted integral while TeX's cmex and *Lucida* offer only a limited set of (2 or 3) sizes. This is true as well for other symbols such as the radical sign, big parentheses, horizontal braces, etc.
- Extending horizontal or vertical strokes does not apply to characters that are not made of single strokes. For example mathematical fonts give the wide hat[5] only a limited set of sizes and no general extension at all. Unexpected (i.e. wrong) results may occur as shown in figure 7.
- There is a gap between some character sizes. *Symbol*, see figure 8 right, offers one right parenthesis with a height[6] equal to 0.864em and offers three symbols to compose

[5] It seems that this symbol is more used in France that in English speaking countries.

[6] We call height of a character the difference $ury - lly$, where ury and lly are respectively the ordinates of the upper right corner and of the lower left corner of its bounding box (see [17, 5.4]); it is given in terms of em, i.e. relative to the current point size.

big right parentheses. The upper part "⎛", the lower part "⎝" and the middle part "⎜". The upper part and the lower part have the same height: 1.220 em. Therefore, the minimum height[7] for a big parenthesis is 2.440 em. In this font the caps height is 0.673 em. While the parenthesis is approximatively 30% higher than the capitals, the minimum big parenthesis is about 330% higher than capitals. With such metrics, it is not possible to get parentheses at the correct size for expressions such as (a_i). Formatters using *Symbol* put extra blanks above and under the a_i; but some formatters draw bad parentheses like "⎛⎞".

- Both *cmex* and *Lucida* fill this gap by offering intermediate medium-size parentheses (figure 8 right).

3 MATH-FLY, A FONT TO HANDLE VARIABLE-SIZED SYMBOLS

Variable-sized symbols have an interesting property: like rules, they are seldomly used at a given size. So, instead of pre-computing all the possible sizes, we think it is preferable to compute them only when they are needed, i.e. just before the character bitmap is copied at the requested position of the page image. Indeed, page description languages like PostScript are capable of producing what we called "dynamic fonts" [19]. PostScript has a cache mechanism that may be disabled: any instanciation of a character results in re-computing the corresponding bitmap. This means that context can be taken into account dynamically[8]. In the case of variable-sized symbols, this context is the bounding box of the expression "inside" this symbol (in terms of embedded structured boxes, e.g. under the upper bar of a radical sign, under a vector-arrow, on the right of an integral sign, etc.).

Math-Fly is a font we are designing to handle mathematical symbols. It is a parametrized font (see section 2.2) and uses analytical values to describe control points. Futhermore these values are context dependant and computed at print time (see [22] for details).

Various classes of problems have to be solved.

3.1 Obtaining outlines

In this first prototype, we are working with public domain fonts and, when needed, products like *Ikarus*, *Fontographer*, etc. are used to digitize hand drawings. In any case, we get outline descriptions with numerical values for each point. Examples shown here are inspired by *Symbol*.

3.2 Analytical computation

Starting with outlines described with numerical coordinates, we have to decide which ones have to be transformed into analytical variables and which values are to be assigned to them.

[7] Actually, this height may be reduced by 20% by overlapping the straight parts of these items.

[8] Such fonts have been used for example to design "lively outlines" [20], or to compute the extension of arabic characters to justify texts [21]. More examples are given in [22,23].

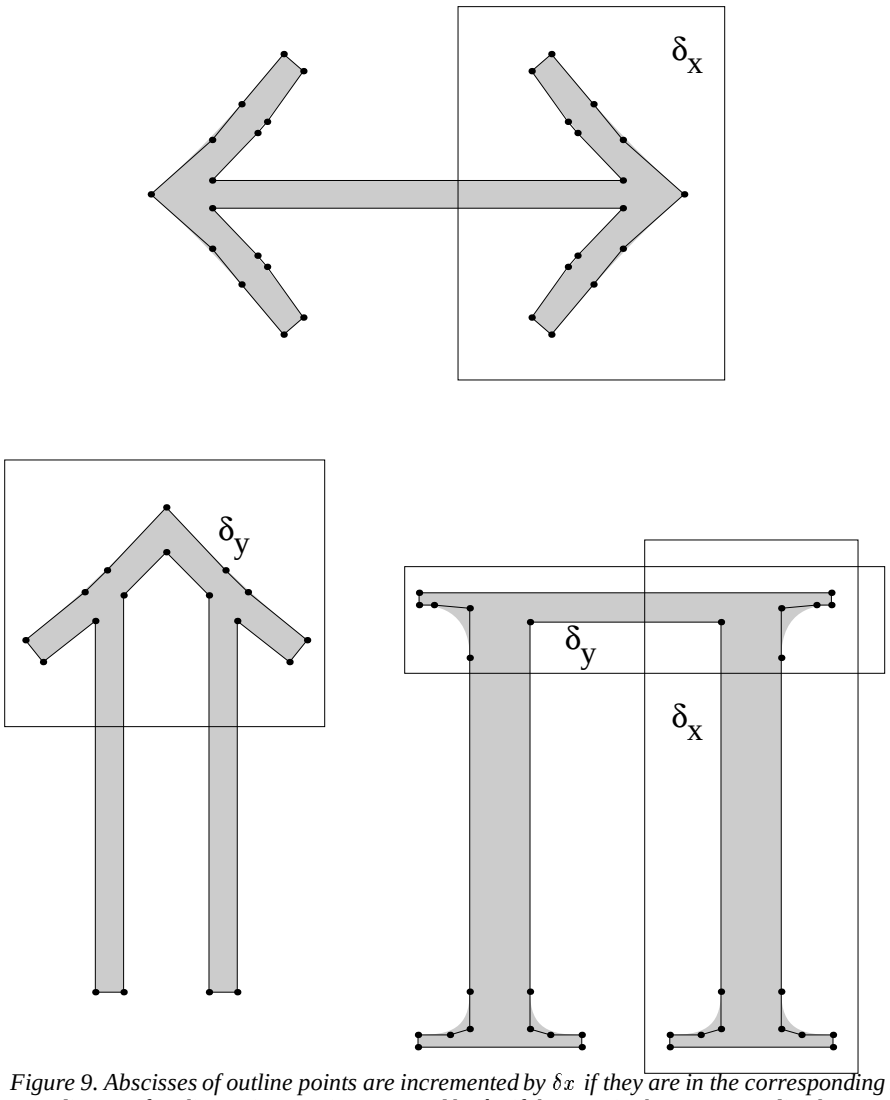

Figure 9. Abscisses of outline points are incremented by δx if they are in the corresponding box; ordinates of outline points are incremented by δy if they are in the corresponding box. Both coordinates are incremented if outline points belong to the two boxes.

3.2.1 Linear symbols

Most of the symbols are made of linear strokes, with outline points only at the ends (junctions, serifs, etc.). Extending such symbols only requires that their ends be globally translated. Horizontal symbol (such as $\Longleftrightarrow$ or $\longmapsto$) abscisses have to be incremented by δx (this value has to be computed by the formatter according to the "content" of this symbol). Vertical symbol (such as $[$, $|$ or $\Uparrow$) ordinates have to be incremented by δy. Some symbols, such as $\sum$, $\prod$ or $\surd$ may support both translations δx and δy (figure 9).

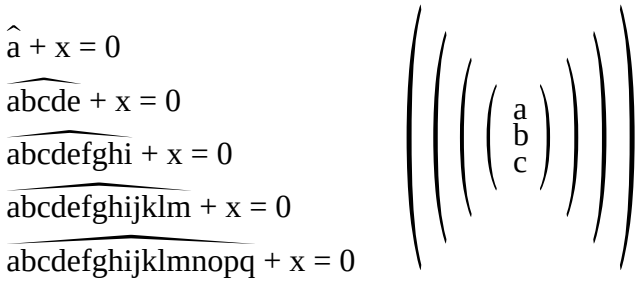

Figure 10. Math-Fly *transforms non-linear symbols such as hats or parentheses having a given direction by performing a slight perpendicular directional change*

3.2.2 Non-linear symbols

The same rule could be applied as well to parentheses, braces, etc. when they are made of linear segments (like in figures 6 and 8). However, if they are curved, a perpendicular deformation is needed as well (see figure 10, right). The same occurs with linear symbols that are not in one direction only, such as hats (figure 10, left).

3.2.3 Optical scaling of large symbols

Outline descriptions of particular symbols, such as integral signs and braces, require Bézier curves intensively. The goal of our font is to increase the size (e.g. the height of an integral sign) without modifying the stem thickness, i.e. by following optical scaling rules.

We are studying[9] the way large metal symbols are designed according to their size. The method is to start from old mathematical symbol collections, such as the braces in figure 4. Different variable sized braces are scanned and their bitmaps are vectorized (we used Agfa *PressView*). Figure 11 shows three parts of braces from figure 4, respectively at 20, 40 and 50 point size. These parts concern only the inner outlines describing the upper bowls. (Hairlines at the end have been dropped because there was too much noise on them due to the scanning process). These three parts have been magnified in the x direction. This figure shows both the outlines and their control points (in gray). It can seen (black lines) that all the sets of corresponding Bézier points are positioned on straight lines. (Small variations are probably caused by very small gouge defects at punch time). Note that these lines are not parallel, like in figure 3.

Other experiments confirm that linear interpolation may be used to define control points. In other words, the same rule applies for large symbols and for letters (see section 2.4.1). So, we propose the following method:

1. Draw a small symbol (e.g. a brace or an integral sign).
2. Draw the equivalent large symbol with the same weight as the small one.

[9] The first results, even if significant, should be confirmed with further important sets of measures on various fonts, either from printed material, from punches or, better, from “smoked proofs”.

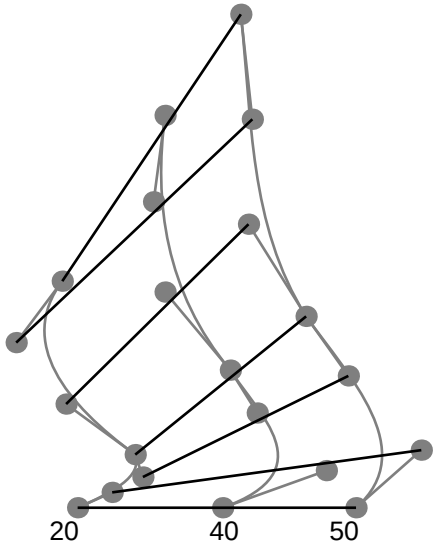

Figure 11. Study of the optical scaling deformation of hot metal braces (from figure 4): Bézier control points are located on straight line segments

3. Use linear interpolation between the corresponding control points.
   Note that these lines may be parallel or not depending on the Bézier curves like in figure 3 (in other words, symbols in the whole do not follow linear scaling).

3.3 Metrics of MathFly

Math-Fly is a true "type 3" PostScript font. However it does not use the same metric as conventional PostScript fonts. When painting a character, three parameters have to be passed to the PostScript `show` instruction: the x, y coordinates of the current point and the encoded value of the character to be drawn. Here we need two more parameters: the expected x and y widths (or what would be the same, the required extensions δ_x, δ_y). Because type 3 fonts are defined as read-only dictionaries, passing parameters (such as δx and δy), use of variables (to keep intermediate results out of the stack) and use of local operators require the use of a dictionary out of the font dictionary.

Furthermore, the standard font metric file has to be modified to give the formatters information on the way symbols are extended. Indeed, a horizontal symbol (like a $\Rightarrow$) cannot be vertically extended while a vertical one (like $\Downarrow$) cannot be horizontally extended. And, more difficult, other information has to be passed to the formatter for small variations (such as the ones for the orientation of parentheses; see figure 10).

Due to the fact that font metrics have their own information, formatters using such a font have to behave on a new basis. This is true as well for Adobe Multiple Masters which requires the Font Metric to be recomputed after the set up. Note that even changing fonts from 8 to 16 bits, like in Unicode, requires rewriting editors [24]. So, it is worthwhile to formulate how an editor can use a font with dynamic metrics.

Let us take an example. Figure 12 shows the formula $A = V_X + V_Y$ where V_X has been drawn with Symbol parentheses and V_Y with MathFly ones. For the left V_X parenthesis,

$$A = \begin{pmatrix} 1 \\ 1 + \frac{1}{\sqrt{2}} \\ X \\ 1 + \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix} + \left(\begin{array}{c} 1 \\ 1 + \frac{1}{\sqrt{2}} \\ Y \\ 1 + \frac{1}{\sqrt{2}} \\ 0 \end{array} \right)$$

Figure 12. The same vector composed with Symbol (left) and with Math-Fly (right) parentheses

the formatter had to work as follows:

1. compute positions x and y of the bottom of the left parenthesis
2. compute its height H
3. compute (using the Symbol metric file) the number of vertical segments to compose it
4. generate PostScript code such as:

```
find font Symbol
scale it at 12 points
moveto x y
show the bottom of the parenthesis
moveto x y+h
show a vertical bar
moveto x y+2h
show a vertical bar
...
moveto x y+H-h
show the top of the parenthesis
```

5. use the Symbol metric file to compute the position of the inner part of the vector (i.e. $x+$ parenthesis width).

For the left V_Y parenthesis, the formatter had to work as follows:

1. compute positions x and y of the bottom of the left parenthesis
2. compute its height H
3. compute, using the MathFLy metric file, the expected δ_x variation
4. generate PostScript code such as:

```
find font Mathfly
```

```
scale it at 12 points
moveto x y
put H and delta x into the font sub-dictionnary
show the left parenthesis
```

5. compute the position of the inner part of the vector (i.e. $x + \delta_x$).

The next section will show how an editor, namely Grif, can do all these computations by using font metric and structured formula.

4 USE OF MATH-FLY FROM GRIF

4.1 Grif overview

Grif is an interactive system for editing and formatting complex documents [25] [26] where documents are represented by their logical structure rather than by their visual aspect. In this respect it is comparable to syntax-driven editors used for editing programs. It is based on SGML Document Type Definitions (DTD), which specify the logical organization of the document to be processed. More precisely, a DTD specifies the types of elements which make up a document and the relationships linking these elements.

All editing commands are performed through a formatted picture of the document displayed on the screen, as in WYSIWYG systems. When modifying the structure or the content of a document, the user acts directly on this formatted picture and sees the results of the commands immediately thereon.

As well as the logical structure, the visual aspect of documents (on the screen or on a sheet of paper) is specified on a generic basis. When defining a new DTD, the user gives *presentation rules* for each type of element defined in the generic logical structure, and Grif uses these rules for building the picture of a document. Thus, the user who edits a document has only to enter its logical structure as well as its content for the system to automatically generate its document picture.

4.2 Abstract Picture

A specific language, called P language, is used to describe presentation rules for each type of element defined in the DTD. Interpretation of these presentation rules is based on the concept of *abstract pictures* [27]. When Grif has a document to print or to display in a window, it first builds an abstract picture which is a high-level description of that document image. This description is device-independent and allows the editor to update the image dynamically in a simple way. In a second step, it translates this abstract picture into the real image which is displayed on the screen or printed on paper sheets. The abstract picture is the interaction support between the application and the user.

An abstract picture is a hierarchy of *abstract boxes*, the concept of box being the rectangle which delimits a document element as defined in TEX [16]. In an abstract picture, boxes are organized as a tree, where each node is a box which encloses all its children. These boxes are termed abstract because all their attributes are defined in a relative way. Let us examine a simple example like the following centered formula:

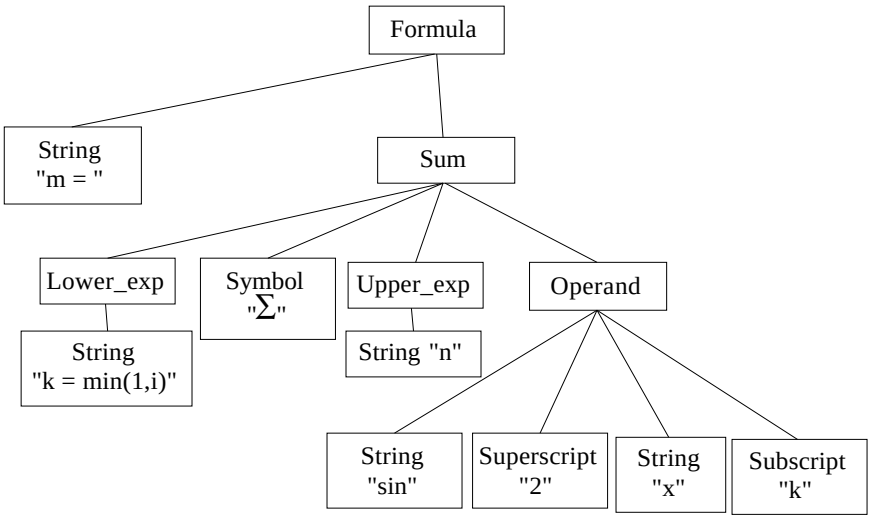

Figure 13. Abstract picture corresponding to formula 1

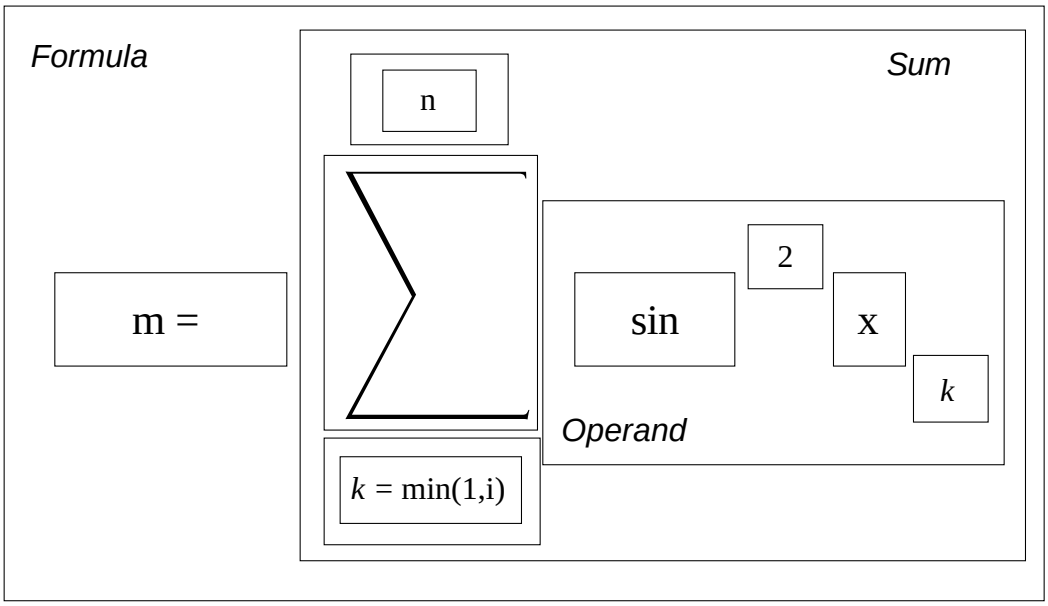

Figure 14. Boxes corresponding to formula 1

$$m = \sum_{k=min(1,i)}^{n} \sin^2 x_k \qquad (1)$$

The abstract picture describing this formula 1 is shown in Fig. 13 while boxes represented by this abstract picture are shown in Fig. 14

As opposed to PostScript [17], this kind of picture description does not locate document elements in a virtual space, but relative to one another. The tree structure gives a first approximation of relative positions which only gives the enclosures amongst boxes. Each node is decorated with position attributes and dimension attributes which express geometric constraints between boxes. These attributes result from constraints expressed in P language.

Therefore each box is located relative to its enclosing box or to one of its sibling boxes, with a fixed distance between two parallel edges or axes of the two boxes. In each direction the position can be defined with reference to a different box. In our example, some position attributes are:

```
Sum:       HorizontalPosition:
                 Left = Previous String.Right;
           VerticalPosition:
                 BaseLine = Previous String.BaseLine;
Symbol:    HorizontalPosition:
                 HorizontalCenter = Lower_exp.HorizontalCenter;
           VerticalPosition:
                 Bottom = Lower_exp.Top;
Upper_exp: HorizontalPosition:
                 HorizontalCenter = Lower_exp.HorizontalCenter;
           VerticalPosition:
                 Bottom = Symbol.Top;
Operand:   HorizontalPosition:
                 Left = Symbol.Right;
           VerticalPosition:
                 BaseLine = Symbol.BaseLine;
....
```

Positions may use the four edges of a box (Top, Bottom, Left, Right), its center and its base line. The base line of a box may be defined relatively to the box itself or to any of its children. In the example, base lines are defined by:

```
String:  BaseLine = Default;
Sum:     BaseLine = Symbol.BaseLine;
Symbol:  BaseLine = Self VerticalCenter + 0.4;
Operand: BaseLine = Child String.BaseLine;
```

The result of the previous set of constraints is that:

- `String` "m =", `Symbol` "$\sum$" and `String` "sin" are displayed on the same baseline, `Symbol` base line being 0.4 below its middle;
- `Upper exp` is displayed centered above the `Symbol` "$\sum$";
- `Symbol` "$\sum$" is displayed centered above the `Lower exp`.

The dimension of a box can also be specified relatively to the dimension of its enclosing box or one of its sibling boxes. The relation then specifies the difference between the dimensions of the boxes, or a ratio between them. A dimension can also be fixed independently of any other box. Examples of dimension attributes are:

```
Symbol:    Width =  Lower_exp.Width;
           Height = Operand.Height * 1.2;
```

This set of constraints express that:

- `Symbol` width depends on the `Lower_exp` width;
- the height of the `Symbol` "$\sum$" is equal to 1.2 times the height of the `Operand`.

The abstract picture permits an incremental display and so ensures high performance for interactive applications such as editors which frequently modify some small parts of pictures. This description enables the application to make the minimum change to the abstract picture.

Tree structure and constraints between boxes offer a powerful mechanism for describing such complex pictures as those frequently found in mathematical formulae. The logical organization of mathematical constructions is described using a DTD and their graphical aspect is specified using the P language. After that, the user just has to manipulate mathematical formulae in logical terms; Grif is responsible for maintaining their real images.

4.3 Output drawing

While the P language allows us to express how to calculate the correct size of mathematical symbols (more precisely the size fo their enclosing box), it is also necessary to have a correct rendering of these symbols. Only *Math-fly* is able to exploit this optimal size evaluation.

Grif computes the abstract picture and its geometrical constraints in order to obtain the correct position and precise size of symbols to print on paper. In the example above, the dimension of the box symbol is `width=lower_exp.width`. Conventional formatters fill this box with the nearest available symbol in the font, i.e. with too small a $\sum$, and add exaggerated blank side bearings, like in formula 1.

In conjunction with the *Math-fly* font metric file, Grif computes the precise extensions δx and δy to be made on the $\sum$ symbol, and passes these values to the font through a dictionary. Formula 1, when handled by Grif, is rendered as shown in figure 15.

$$m = \sum_{k=min(1,i)}^{n} \sin^2 x_k$$

Figure 15. Formula 1 as seen by Math-fly + *Grif*

Programming these modifications was possible thanks to two mechanisms: ECF (External Call Facilitites) and API (Application Programming Interface) [28].

4.4 Remaining problems

MathFly is still in progress. Here are some yet unsolved problems.

- Due to the fact that mathematicians have recently become used to seeing composite symbols such as light parentheses like in (figure 12 left), there is no guarantee that parentheses with the same boldness as radical signs ((figure 12 right) will be accepted. Futhermore, is an extended summation sign (figure 15) better or worse than one with blank spaces (formula 1)? We do not have an obvious answer today.
- Our "type 3" MathFly font does not support "type 1" hints mechanism. And type 1 fonts do not allow parameters to be passed easily. Encapsulation concepts would be welcomed in Page Description Languages.
- Today, bitmapped fonts are used on screens which do not allow dynamic evaluation of curves. Straight lines could be used to simulate these curves.

5 CONCLUSION: FONT EVALUATION TECHNIQUES

Three different ways of making digital typefaces have been described in this paper. They mainly differ as to the time when the character bitmaps are computed:

1. METAFONT enables arbitrary relations to be defined between parts of a character. By changing parameters, Knuth defined, in a single code, all body-dependent variations of the font named Computer Modern. However, METAFONT works in three steps: 1) Before using any printer, screen, or typesetter, it produces bitmaps for a given point size, e.g. Computer Modern Roman at 5 point size. 2) This bitmap is loaded into the printer. If a text needs Computer Modern at n different point sizes, all of these n corresponding bitmaps have to have been loaded (and so computed) before printing starts. 3) When a character is needed, the bitmap is copied into the page.
2. Multiple Masters works in three steps: 1) The font is loaded into the printer as a set of parametrized fonts. 2) During the initial "set up", all the potential PostScript fonts[10] are built (e.g. Times at 6 point size, Times at 7 point size, etc.). 2) Each font is used as a regular PostScript font: character bitmaps are computed only once (and saved into a cache memory at the first occurrence of the concerned character) and copied into the page.
3. Dymanic fonts work in two steps: 1) The font is loaded into the printer as a set of parametrized fonts. 2) When a character is needed, its bitmap is computed according to the actual parameters and copied into the page.

Table 1 exhibits the fact that bitmap evaluation behaves like passing parameters in traditional programming languages.

This concept of parameter by name (i.e. of a parameter that is a procedure) is not obvious. This may be the reason why dynamic fonts are not very well understood either.

Another point of interest is to compare the different shapes of a letter at different body sizes (like in figure 1-bottom) with the different images of a cartoon: two extreme images are given and and the intermediate shapes are automatically computed with no user interaction. This operation is known as shape averaging, shape interpolation, metamorphosis and shape

[10] Note that Multiple Master fonts enable other variations than optical scaling, e.g. variations on boldness, serifs shape, etc.

Table 1. Comparaison between passing parameters and computing bitmaps

Parameters :	by value	by reference	by name
Evaluation :	constant	once	at each occurrence
Bitmap :	METAFONT	Mutliple Masters	Dynamic fonts

blending. There, chaotic intermediate shapes can be avoided [29]. This is not yet the case for fonts.

Acknowledgements

Our thanks go to the Opera team where Grif has been designed and especially to Vincent Quint for his advice and Michel Jollivet for his help in implementing MathFly through the API mechanism. Special thanks are due to the anonymous reviewers of a previous version of this paper and to the EPODD team for their help for the final edition.

REFERENCES

1. Mark Jamra, 'Some elements of proportion and optical image support in a typeface', in *Visual and Technical Aspects of Types*, ed., Roger D. Hersch, 47–55, Cambridge University Press, Cambridge, UK, (1993).
2. Richard Southall, 'Character description techniques in type manufacture', in *Raster Imaging and Digital Typography II*, eds., Robert A. Morris and Jacques André, pp. 16–27, Cambridge, UK, (October 1991). Cambridge University Press.
3. Richard Rubinstein, *Digital Typography, An Introduction to Type and Composition for Computer System Design*, Addison-Wesley Company, Reading, USA, 1988.
4. *Visual and Technical Aspects of Type*, ed., Roger D. Hersch, Cambridge University Press, Cambridge, UK, 1993.
5. Peter Karow, *Digital Typefaces*, URW Verlag, Hamburg, Germany, 1994.
6. Mark Agetsinger, 'Adobe Garamond: a review', *Printing History*, **26–27**, 69–98, (1991–1992).
7. Jonathan Seybold, 'Adobe's 'MultiMasters' technology: Breakthrough in type aesthetics', *The Seybold Report on Desktop Publishing*, **5**(7), 3–7, (March 4 1991).
8. Adobe Developer Support, *Adobe Type 1 Font Format: Multiple Masters Extensions*, 1992.
9. Arlene E. Karsh, 'Composition quality: Can URW 'one-up' Gutenberg with *hz*-program?', *The Seybold Report on Publishing Systems*, **22**(11), (February 22 1993).
10. Peter Karow. *hz*-program, microtypography for advanced typesetting. URW, Hamburg, germany, 1992.
11. Donald Knuth, 'Tau epsilon chi, a system for technical text', report STAN-CS-78-675, Stanford Computer Science, (september 1978). Now appears as [16] and [30].
12. Bridget Lynn Johnson, *A Model for Automatic Optical Scaling of Type Designs for Conventional and Digital Technology*, Master's thesis, School Printing in the College of Graphic Arts of the Rochester Institute of Technology, May 1987.
13. Donald Knuth, *Computer Modern Typefaces*, Addison-Wesley, Reading, MA, 1986.
14. Yannis Haralambous, 'Parametrization of PostScript fonts through METAFONT – an alternative to Adobe Multiple Master fonts', *Electronic Publishing: Origination, Dissemination, and Design*, **6**(3), 145–157, (September 1993). RIDT'94 proceedings.
15. Michel Goossens, Frank Mittelbach, and Alexander Samarin, *The LaTeX Companion*, Addison-Wesley, Reading, MA, 1994.
16. Donald Knuth, *The TeXbook*, Addison-Wesley, Reading,MA, 1984.
17. Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley, Reading, MA, 1991. second edition 1993.

18. Chuck Bigelow and Krist Holmes, 'The design of Lucida: an integrated family of types for electronic literacy', in *Text Processing and Document Manipulation*, ed., J.C. van Vliet, pp. 1–17. Cambridge University Press, (1986).
19. Jacques André and Bruno Borghi, 'Dynamic fonts', in *Raster Imaging and Digital Typography*, eds., J. André and R.D. Hersch, pp. 198–203. Cambridge University Press, (1989).
20. Erik van Blokland and Just van Rossum, 'Different approaches to lively outlines', in *Raster Imaging and Digital Typography II*, eds., R. Morris and J. André, pp. 28–33. Cambridge University Press, (1991).
21. Johny Srouji and Daniel Berry, 'Arabic formatting with *ditroff/ffortid*', *Electronic Publishing – Origination, Dissemination and Design*, **5**, 163–208, (December 1992).
22. Jacques André and Irène Vatton, 'Contextual typesetting of mathematical symbols taking care of optical scaling', Research report 1972, Inria, (October 1993).
23. Jacques André. *Création de fontes en typograhie numérique*. Mémoire d'habilitation à diriger les recherches, Université de Rennes, 29 September 1993.
24. Chuck Bigelow and Kris Holmes, 'The design of a Unicode font', *Electronic Publishing: Origination, Dissemination, and Design*, **6**(3), (September 1993). RIDT'94 proceedings.
25. Vincent Quint and Irène Vatton, 'Grif: an interactive system for structured document manipulation', in *Text Processing and Document Manipulation*, ed., J.C. van Vliet, pp. 200–213. Cambridge University Press, (1986).
26. Richard Furuta, Vincent Quint, and Jacques André, 'Interactively editing structured documents', *EPODD, Electronic Publishing – Origination, Dissemination and Design*, **1**(1), 19–44, (april 1988).
27. Vicnent Quint and Irène Vatton, 'An abstract model for interactive pictures', in *Human-computer interaction, Interact'87*, eds., H.-J. Bullinger and B. Shackel, pp. 643–647. North-Holland, (September 1987).
28. Vincent Quint and Irène Vatton, 'Making structured documents active', *Electronic Publishing – Origination, Dissemination and Design*, **7**(1), (1994).
29. Thomas W. Sederberg and Eugene Greenwood, 'A physically based approach to 2-D shape blending', *Computer Graphics*, **26**(2), 25–34, (july 1992).
30. Donald Knuth, *The* METAFONT*book*, Addison-Wesley, Reading, 1984.