Justify Just or Just Justify

Mohamed Elyaakoubi

Azzeddine Lazrek

Journal of Electronic Publishing

Volume 13, Issue 1, Winter 2010

DOI: http://dx.doi.org/10.3998/3336451.0013.105 [http://dx.doi.org/10.3998/3336451.0013.105]

Abstract

This paper describes a formalism for the justification of texts written in an Arabic alphabet-based script, within some approved calligraphic rules, that would produce better typographical quality than current publishing systems. Specifically, we improve the optimum-fit algorithm by taking into account the existence of allographic variants and stretched forms with kashida *(a feature in some cursive alphabets) provided by the font. This sophisticated algorithm breaks the paragraph into lines in an optimal way; it does not just justify each line. Thus, it allows selecting the optimal version among several variants. This approach could be extended for the composition of multilingual texts.*

Introduction

Typography needs to be considered as much in electronic publishing as in book production (Châtry-Komarek 2003; Peck 2003), because both books and e-texts should be readable: the material must be designed to be legible and to communicate meaning as unambiguously as possible. Among their tools, graphic designers need typographical skills as well as design skills for both books and websites. Design is not by itself sufficient to convey a message to book readers or Website visitors; the quality of the text is just as important. A reader should be assisted in navigating through text with ease, using optimal inter-line, inter-letter, and inter-word spacing and text justification, coupled with appropriate line length and position on the page.

One measure of good typography is *typographical gray,* the appearance of the page when it is held so far away that characters are not legible. Typographical gray, also known as the *color of the text,* is the black area on a white sheet, the impression produced on the eye by the global vision of a page of text. This impression, interestingly, has a significant influence on legibility. The most uniform typographical gray is the easiest on the eye, and makes reading the text easier (Zachrisson 1965; Comber 1994). But achieving a uniform typographic gray is not easy.

Of course, the typographical gray depends on the direction of the script and its environment. A left-aligned text, also called *flush left*, provides a ragged-right edge. Text centering produces ragged edges on both sides of the text, and justification balances the margins on both sides of the text. No alignment is necessarily better than the others; it depends on what the text looks like when it is aligned. Designers for traditional publishers choose their alignment (left, centering, or justification) by taking into consideration how the text will be perceived by the human eye.

Typically, book publishers (and many Web publishers) prefer fully justified text, aligned on both the right and left sides. Most books are justified using automatic justification programs that are part of typesetting systems. These programs depend on algorithms that insert spaces between words to get the margins even. Automated systems are not perfect, however, and obtaining a uniform typographical gray has never been easy. The main reason for this is the impossibility of achieving *equal* inter-word spacing in different lines. The rivers and the alleys produced by the random disposition of spaces in lines in a justified text can produce an uncomfortable effect for the reader, because the irregular spaces catch a reader's eye. Without appropriate algorithms for fully justified text, unjustified or ragged-right settings might sometimes be preferable.

Today's professional justification programs use finer and finer resolutions, micro-typography and delicate typographic rules governing design of the fonts themselves, the typefaces, and the spaces between *glyphs*,[1] [#N1] as well as macro-typography that considers the text as a block on its own, and looks at the overall structure of the page (Hochuli and Kinross 1996).

"Most people trust their software to automatically lay out their pages, resulting in typographical anarchy."

Such professionalism is not available to everyone with a desktop-publishing system, a Web site, or a word-processing program that controls the appearance of text. Yet to a certain extent, typography has become a job for everyone, with or without training. Most people trust their software to automatically lay out their pages, resulting in typographical anarchy.

This paper seeks to introduce non-professional publishers and typesetters to some of the elements of typography, particularly justification issues.

Breaking paragraphs into lines

A typesetting system has to break paragraphs into lines. A line break is usually placed at a word boundary or after a punctuation mark. It can also occur following a hyphen.[2] [#N2] Sometimes, a break between two words is not desirable: for instance, within a full name of a person or inside a compound word. In such situations, the word split can usually be avoided if the typesetter uses a hard, non-breaking space. However, typesetting systems cannot automatically recognize where non-breaking spaces are needed because most of them cannot recognize the semantics of the text. Instead, they break paragraphs into lines by using optimization algorithms based on various criteria. Some are discussed below.

Greedy algorithm

An easy way to break a paragraph into lines is to use the greedy algorithm. This algorithm basically puts as many words on the line as it can contain, repeating the process for each line until there are no more words in the paragraph. The greedy algorithm is a line-by-line process. Each line is handled independently. The following pseudo-code implements this algorithm for left-to-right texts:

```
SpaceLeft = LineWidth
for each Word in Text
     if Width(Word) > SpaceLeft
          insert LineBreak before Word in Text
          SpaceLeft = LineWidth - Width(Word)
     else
          SpaceLeft = SpaceLeft - (Width(Word) + SpaceWidth)
```

Where `LineWidth` is the width of a line, `SpaceLeft` is the remaining width of space on the line to fill, `SpaceWidth` is the width of a single space character, `Text` is the input text to iterate for each line, and `Word` is a word in this text. This algorithm is very simple and fast, and it puts the words on a minimum number of lines. It is used by many text processors, including Open Office and Microsoft Word.

Optimum-fit algorithm

In the optimum-fit algorithm, justification of some lines is sacrificed to improve the justification of other lines, even those that may not follow immediately. The main goal is to get an optimum result for the whole paragraph.

For example, the optimum-fit algorithm might have to break the sequence `xxx xx xx xxxxx` over a line whose width is `xxxxxx`. The greedy algorithm would break this list of words as follows:

```
xxx xx
xx
xxxxx
```

The result is a long line followed by a short one, and a rather unpleasant pair of right edges. In contrast, the same paragraph processed with the paragraph-based line-breaking optimum-fit algorithm results in a better right margin:

```
xxx
xx xx
xxxxx
```

The key idea behind such a paragraph-based algorithm is to use dynamic programming to globally optimize an aesthetic cost function (Knuth and Plass 1981). The strategy adopted in the optimum-fit algorithm is to find a break opportunity at a given distance from the beginning of the line. The algorithm typically finds several candidate points for breaking the line. For each line break, a cost function e_i and error on the line is defined. If the line is perfect, $e_i = 0$; if a line composition is not possible, e_i will be ∞. The algorithm then seeks the best arrangement to minimize the total errors $e = \sum e_i$ for all the lines in a paragraph.

This principle was first introduced by Donald E. Knuth and used in $\TeX$ (Knuth 1986). This algorithm will be discussed in more detail below.

Justification in Latin typography

Spacing

The algorithms that break paragraphs into lines are each designed to search for an optimum approach in different ways. Typically they adjust the spaces between words and even between characters. The goal is to obtain lines that end on the right margin.

To give a line the necessary flexibility, a space between two words is not always changed into a space character. Some of these spaces are transformed into line ends. Others are transformed into triple values `(id, max, min)`. Thus, spaces have a normal size `id` that can be stretched up to a `max` or shrunk to a value `min`. If necessary, a space's value beyond `max` can be tolerated, but the space's value can never be less than `min`.

The logic of the spacing is:

1. Put the words on the line, word after word.
2. Continue until a word comes at the end of a line and overflows onto the margin.
3. Try to shrink inter-word spaces without going less than `min`.
4. If that is not possible, try to stretch inter-words spaces without exceeding `max`.
5. If that is not possible, stretch while exceeding `max`.

Hyphenation

To obtain e-documents with visually homogeneous paragraphs, the spaces between words should be minimized. Hyphenation improves the text's alignment by reducing the space between the last word of the line and the right margin. A word that is too long to fit at the end of a line is then broken, typically between syllables.

The logic for hyphenation is the same as for spacing through step 4, and then:

5. If that is not possible, try to hyphenate.
6. If that is not possible, stretch while exceeding `max`.

A value called *badness* can be associated with each possible line break. This value is increased if the space between the words must stretch or shrink too much to make the line the correct width. A best-fit strategy can help in deciding among shrinking, stretching, and hyphenation, in order to minimize the value of badness.

In the past, the hyphenation was done manually, often based on a hyphenation dictionary that identified hyphen locations. An elaborated method find nearly all of the legitimate places to insert hyphens in words in Latin-based languages (Liang 1983). This method is based on an organized tree structure of tries: a particular class of tree structure, the term was derived from the word *retrieval* (Fredkin 1960). The tree contains a list of hyphenation patterns, combinations of letters that allow or prohibit word-breaking. The method assigns priorities to breakpoints in letter groups. The patterns reflect the hyphenation rules of a language, so there are as many pattern tree structures as there are languages. This algorithm was implemented by Donald E. Knuth in TEX.

Other practices

Gutenberg’s typography

Five hundred years ago, Johannes Gutenberg struggled to reach, in typesetting, the same quality as in contemporary handwritten manuscripts. By hand, words can be written tightly or loosely without getting too many dark spots or too many clear spots in the proof. Gutenberg used and improved existing techniques (Wild 1995). When he typeset the 42-line Bible, he used types of various widths, and he employed ligatures and abbreviations so that he could get the right justification.

Nowadays, we can imagine how his typesetting works. When he set a line, he tried types of variant glyphs with various widths for the same characters until the line had the right length. Gutenberg also used abbreviations and ligatures freely to make line justification easier. Those approaches were similar to what copyists were doing at the time, but those techniques are now rarely used in text processing and publishing systems. To use Gutenberg’s techniques successfully, the presence or absence of ligatures (links between letters) or glyph variants must not change the reader’s experience. Today, in Latin text, the only commonly used ligatures are the f-ligatures (fl, ff, fi, ffi, ffl).

Horizontal glyph scaling

Hz-program: The hz-program is software for advanced typography. It was designed by Hermann Zapf in collaboration with URW.[3] [#N3] Its main goal was “*to produce the perfect gray type area without rivers or holes with too-wide word spacing*” (Zapf 1993). Zapf describes his program as a complete aesthetic program for micro-typography. In the beginning, he wanted to match, through computers, the level of justification reached by Gutenberg in typography—a quality that has been considered unreachable in the following 550 years (Karow 1997).

The hz-program strategy is partly based on a typographically acceptable stretching or shrinking of letters, called *scaling*. In addition, a kerning program calculates kerning values. The kerning is not limited to negative changes of space between two critical glyphs; in some cases it allows positive kerning, which adds space on the line. Zapf ignored Gutenberg’s ligatures and abbreviations to make the typesetting both easier to do and easier to understand. The abbreviations of the 15th century are not familiar today.

The hz-program is not available for purchase, testing or reviewing, and no information about its implementation is available; it is URW's black box.

Improved optimum-fit: In his dissertation, Hàn Thé Thành (1999) attempted to improve the quality of the typographical gray produced by the optimum-fit algorithm. The idea is based on adapting the spaces between words once the paragraphs have been broken into lines. Instead of changing the inter-word spacing only, he expands the fonts on the line slightly as well. This approach can minimize excessive stretching of the inter-word spaces. Thành's ideas were inspired by the hz-program. He attempted to closely control the limits where such manipulations are allowed. He set parameters for better control of the acceptable stretching or shrinking of glyphs. This width modification is implemented through a horizontal scaling of fonts in pdfT$_E$X.

Reactions

The main contributions of Zapf and Thành were to bring Gutenberg's level of high-quality typography into digital typesetting. They went back to the earliest days of printing for their standards. They found that a font's size variation helps to improve the inter-word spacing and therefore to make the grayness on the page more uniform. However, it must be used with great care. Creating rules for the use of the size variation is not easy. A close examination of the Gutenberg 42-line Bible shows that he only cuts off some corners/noses from his black-letter types. Gutenberg kept the counters—the space between the vertical strokes of letter n, for instance—the same. Figure 1 shows the normal forms above, and the narrower forms—which lack the corners of the upper types on their left-hand side—below. The narrower forms were meant to be placed to the right of the letters that have such corners on their right hand side. In this way, the letters could be placed closer to each other, and some space could be saved. For this reason, Gutenberg's narrower forms hardly stand out as different in the text. In any case, the inter-word spaces are not replaced with spaces within the glyphs' counters through horizontal scaling of glyphs.

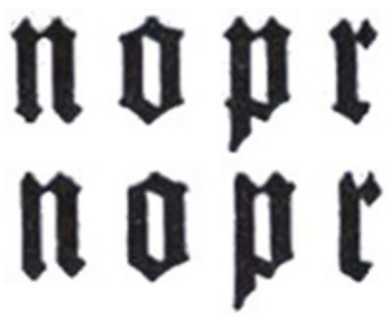


Figure 1:

Cutting corners of typefaces

The width variation of glyphs cannot be used just anywhere, but only for some selected glyphs

(Thành 1999). For example, Peter Karow stated that the letter `i` could not be made wider or narrower in the glyph scaling of the hz-program (Karow 1997).

Glyphs may be bold or slim (see figure 2), making lines stand out. The width modification of glyphs through horizontal scaling should be approached very carefully and parsimoniously.

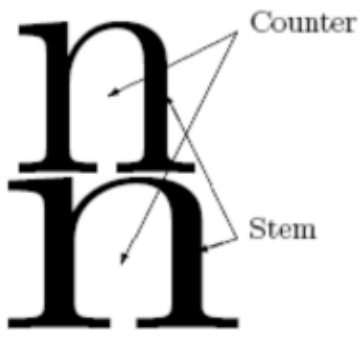


Figure 2:

Horizontal scaling of glyphs

Breaking the vertical list into pages

After the paragraph has been broken into lines, a typesetting system places the lines onto the current vertical list, also known as *the current page*. Each line of the paragraph is a horizontal box. In typesetting, a vertical list can contain vertical glue and vertical penalties. A line-breaking algorithm places vertical glue between the lines, so that the distance between *baselines*[4] [#N4] is uniform. A line-breaking algorithm also inserts vertical penalties (or rules against breaks) between the lines, to assist the page-breaking process. For example, there is usually a penalty for a page break immediately after the first line of a paragraph, or a penalty for a page break immediately before the last line of a paragraph.

Look-ahead problem

The line-breaking algorithm “looks ahead” before it makes any decisions as to where the first, or any other, line break occurs. Each line break is considered not by itself but in the context of the other line breaks. The page-breaking algorithm does not carry out such a look-ahead. Each page break is considered in isolation, without regard for the other pages in the document.

The line-breaking algorithm produces lines of text. These lines are then placed on the main vertical list. If enough material has collected, the page-breaking algorithm cuts off the material for one page, and the output routine is called.

The page-breaking algorithm can use the idea underlying the line-breaking algorithm to improve its look-ahead performances (Fine 2000).

Justification in Arabic typography

Arabic writing has different characteristics from Latin writing. Some of these characteristics contribute to the processing complexity of the justification algorithms that can be used in Arabic. In particular, Arabic writing is cursive in its printed form as well as in its handwritten one. The letters change according to their position in the word, according to the surrounding letters, and in some cases according to the word's meaning (ex. The word *Allah* (God), and the word *Mohamed* when it means the prophet Mohamed (see figure 3)). The alternative positions are then interdependent. The exit point of each glyph is attached to the entry point of the following glyph and there is no hyphenation.

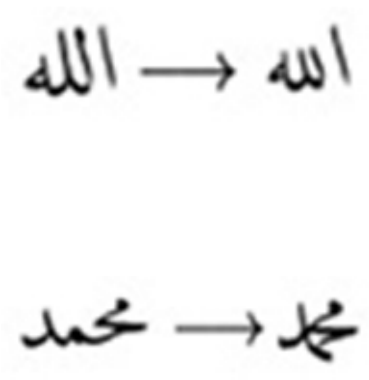


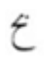
Figure 3:

Special morphology

Ligatures

The cursive nature of Arabic writing implies, among other things, a wide use of ligatures (Haralambous 1995). Indeed, Arabic writing is rich in ligatures. Some of them are mandatory and obey grammatical and contextual rules (Haralambous 1995). Others are optional and exist only for aesthetic reasons, legibility, or justification. Moreover, the connection of letters can lead to the introduction of implicit contextual ligatures. An explicit ligature is the fusion of two, three, or even more graphemes.

Generally, the ligature's width is less than the one of a fused graphemes group. The aesthetic ligature



(right) in figure 4 is 9.65 pt wide, whereas the ligature in the simple contextual substitutions (middle) is 14.75 pt wide.

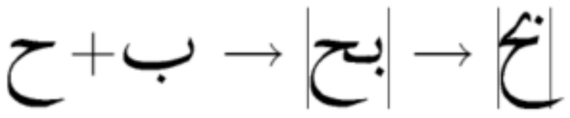


Figure 4:

Contextual and aesthetic transformations

Controlling the ligatures' behavior by converting the implicit ligatures into aesthetic ones gives some flexibility to adjust the word for the available space on the line. The example in figure 5 shows three ligature levels: mandatory simple substitutions, aesthetic ligatures of the second degree, and finally, aesthetic ligatures of the third degree. The last two ligature levels provide different possibilities for shrinking of the same word.

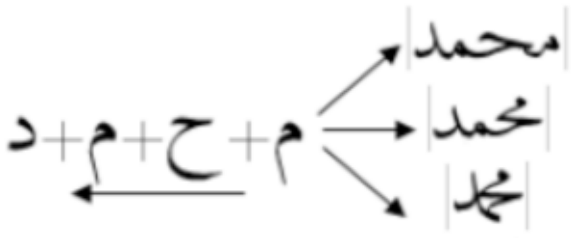


Figure 5:

Various levels of ligatures

The use of second and third degree aesthetic ligatures has to take into consideration the constraints of legibility. A typesetting system should take into account three options for ligatures. For the first level, there are only implicit contextual ligatures. This level also covers mandatory grammatical ligatures, such as the Lam-Alef ligature; it is recommended for textbooks or books for the general public where it is necessary to avoid confusion between letters and reading ambiguities. In a second-level publication, it is possible to take some liberty and use some aesthetic ligatures. In the third level, the use of aesthetic ligatures of higher degrees is allowed, and graphic expressions are free.

The decision to use explicit ligatures to improve justification must take into account the graphic environment and the block regularity of the text. In calligraphy, once an aesthetic ligature is used, there is no obligation to use this ligature in all the text occurrences. The justification problem can be sorted out with *kashida* in texts composed only by implicit ligatures.

The use of ligatures to justify lines is not limited to Arabic writing. Adolf Wild, conservator of the Gutenberg Museum in Mainz, examined the Gutenberg Bible from a typographical point of view and determined that at the lines level, in some cases, Gutenberg justified the text through using ligatures instead of using the variable spaces we are familiar with today. (Wild 1995)

Kashida

The connections between Arabic letters are curvilinear bridges. They are extensible. This property —called *kashida*, *tamdid*, *madda*, *maT*, *tTwil*, or *aliTalat*, etc.—is a feature of Arabic script that is rarely met or maintained in other writing systems (see figure 6). Kashida is used in various circumstances for different purposes:

- *emphasis:* stretching to emphasize an important word or to correspond to phonetic inflection;
- *legibility:* to find a better letter layout on the baseline, and to correct the cluttering that can appear at the joint between two successive letters in the same word;
- *aesthetics*, to embellish a word;
- *justification*, to justify a text.

Kashida is not a character in itself. It allows stretching some letter parts while its body is kept rigid. The example in figure 6 shows compositions of the Arabic word Mohamed. The arrows indicate the use of kashida, in two extensibility levels.

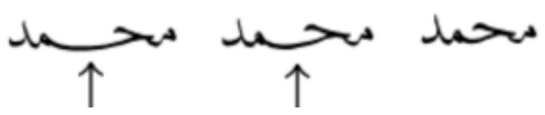


Figure 6:

Various curvilinear kashida

There are three kinds of stretching: mandatory, allowed, and prohibited. The typographical strength of a text can be determined, among other factors, by whether it respects mandatory stretching and/or eschews prohibited stretching.

In terms of Arabic text justification, kashida is a typographical effect that allows lengthening letters in some carefully selected points on the line within determined parameters so that the paragraph can be justified. The Arabic term *tansil* refers to selecting good places to insert kashida.

Current typesetting systems: In terms of text processing tools, curvilinear kashida is, generally, still beyond what the majority of typesetting systems can offer. As we have seen, kashida is not a character in itself, but an elongation of some letter parts. To implement kashida, the majority of typesetting systems proceed by inserting rectilinear segments between letters. The resulting typographical quality is unpleasant. Due to the lack of adequate tools, the solution consists of inserting a glyph. That is, rather than computing parameterized Bézier curves in real time, a ready-to-use glyph is inserted. Moreover, whenever stretching is performed by means of a parameterized glyph coming from an external dynamic font, the current font context is changed.

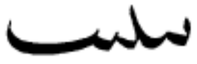
Curvilinear extensibility of glyphs can be offered through the a priori generation of curvilinear glyphs for some predefined sizes. Beyond these sizes, the system will choose curvilinear primitive and linear fragments. Of course, this will violate the curvilinear shape of letters and symbols if they are extended greatly.

A better approach consists of building a dynamic font (Berry 1999; Lazrek 2003; Sherif and Fahmy 2008; Bayar and Sami 2009) through parameterizations of the composition procedure of each letter. To handle the elongations, a letter is decomposed into two parts: the static body of the letter and its dynamic part, capable of stretching. The introduced parameters indicate the extensibility size or degree.

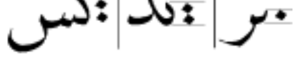
Allographs

Essentially, there are up to four different shapes for each letter in Arabic: *isolated*, *initial*, *median,* or *final* form. Allographs are the various shapes that a letter can have while keeping its place in the word. For instance, the initial form of the letter *Beh* can have—in the same calligraphic style—more than one shape. *Allometry* is the study of the allograph phenomenon, shape, position, context, etc. Generally, allographs are chosen by the writer for aesthetic reasons. However, in Arabic calligraphy, sometimes an allograph is desired and even recommended. The shapes of letters may change according to the nature of the neighboring letters, and in some cases according to the presence of kashida. Some rules concerning use of the allograph are:

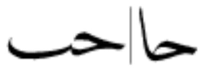
- the shape of the median form of the letter Beh should be more acute when it comes between two spine letters:



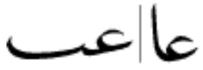
- the initial form of the letter Beh can take one of three allograph shapes, determined by the letter that follows:



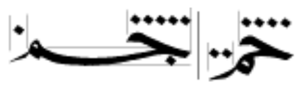
- the initial form of the letter Hah should be a *lawzi* Hah if it precedes an ascending letter:



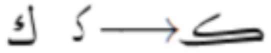
- the initial form of the letter Ain should be a *finjani* Ain if it precedes an ascending letter:



- the initial form of the letter Hah, as well as the final form of the letter Meem, change their morphologies in presence of kashida:



- the letter Kaf changes its morphological shape in case of stretching and should be changed into *zinadi* Kaf:



Back to Zapf and Thành

Adapting the approaches used to improve the justification of Latin text to Arabic typography does not seem to be the best solution. Zapf and Thành reasoned on the basis of Latin typography, where there are individual glyphs and no tools similar to kashida that can be used to justify lines. Indeed, kashida is not a simple horizontal scaling to enlarge letter width. In some cases, operation of kashida on the letter can totally change its glyph's morphological shape (see figure 7). The use of kashida is governed by rules and customs inspired by manuals and treatises on Arabic calligraphy (Benaatia, Elyaakoubi, and Lazrek, 2006).

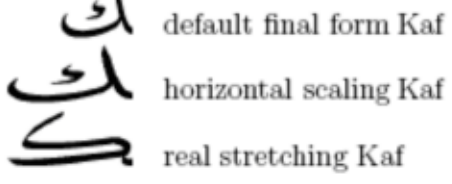


Figure 7:

Arabic stretching letter

Diacritic marks

A diacritical mark is a sign added to a letter, like the acute accent on the letter e which produces é. Diacritics are placed above or below letters, or in some other position such as within the letter or between two letters. In different scripts, diacritical marks have common phonetic and linguistic roles: they change the sound value of the letter to which they are added. However, in other alphabetic systems [http://en.wikipedia.org/wiki/Alphabet#Types] , diacritics may perform other functions. For example, Arabic vocalization marks indicate short vowels applied to consonantal base letters. This vocalization is sometimes omitted altogether in writing.

Arabic diacritical marks have an additional typographical role, which is to fill the void produced by position and juxtaposition of letters on the line. This task is influenced by the effects of Arabic text justification. If kashida is used to manage Arabic lines, it influences the positioning and the measurements of the Arabic diacritical marks (Hssini, Lazrek, and Benatia 2008) (see figure 8). Additionally, the presence of diacritical marks either above or below glyphs adds a vertical inter-line space that should be taken into account in vertical adjustments. Thus, the vertical document adjustment is more delicate.

In this paper we are considering text without taking into account the effect of vocalization, and we have chosen only to address horizontal justification, without considering vertical justification, which is also an issue in Arabic.

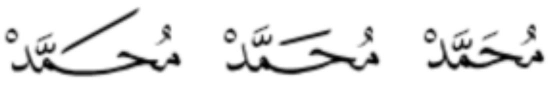


Figure 8:

Diacritical marks with various sizes

Optimum-fit

For about 30 years the paragraph builder based on the Knuth's algorithm—called *optimum-fit*—has been the only method for justifying a paragraph. It is not a simple line-by-line justification. It has the advantage that all the lines of the paragraph are taken into account in the regulation of the spaces. Therefore, the result is rather homogeneous. The optimum-fit algorithm has been implemented by Adobe to regulate text blocks in InDesign software. This algorithm is based on three simple primitive concepts called *boxes*, *glue,* and *penalties*, and avoids backtracking through a judicious use of the techniques of dynamic programming. The purpose of this section is to give an overview of the part of Knuth's idea that is especially relevant to this article.

Glue/penalty model

In the glue/penalty model, a glyph or a ligature is represented by a box with fixed dimensions (*width*, *height*, *depth*) while spaces, or their equivalents, are represented by glues. The concept of glue is more general than that of space. Indeed, a glue is not an *empty* box with a fixed width; it can be stretched or shrunk. Therefore, in addition to its natural width, it has two more attributes: its stretchability and its shrinkability. The glues do not become wider or narrower in the same way as a normal space, but in proportion to their stretchability and shrinkability. For example, according to the English and German typographic traditions, the inter-word space should be increased whenever it comes after a punctuation mark ending a sentence. The glue in such situations should have more stretchability than a space between two words in the same line. The composition weakness of a text is determined, among other things, by the quality of its spacing. In a good composition, the glues approach their natural sizes. Meanwhile, the badness reflects the amount of space at the right margin.

A penalty is an element that we can insert in a horizontal list representing the paragraph to discourage or encourage the typesetting system to break the list at that place. A positive penalty indicates a bad breakpoint, whereas a negative one indicates a good breakpoint. A positive infinite penalty prevents breaks and a negative infinite penalty forces breaks. A penalty is associated with each breakpoint, even if this penalty is null. However, in some situations the composition presents an aesthetic cost. Consequently, there are many types of penalties. For example, the presence of hyphenation adds a non-null penalty. If two consecutive lines end with hyphen breaks, or if they are visually incompatible, positive penalties are added. A value measuring the composition quality is calculated according to glue badness and whole penalties are assigned to the current breakpoints or the current lines. The computed value is called *demerit*.

Algorithm

Considering the example of the section "Optimum-fit algorithm", we demonstrate that the result given by the optimum-fit is more optimal than the result given by the greedy algorithm. The first one optimizes a square of the remaining spaces, also called *badness of lines*, to produce a more aesthetically satisfying result.

In this example, the horizontal list to break is: `xxx xx xx xxxxx`, for a line width `6`. For simplicity, we consider a fixed-width [http://en.wikipedia.org/wiki/Fixed-width] font. If the cost of a line is defined by the square of remaining space, the greedy algorithm would give a sub-optimal solution for the problem, and would look like this:

```
xxx xx          badness=0, cost=0
xx              badness=4, cost=16
xxxxx           badness=1, cost=1
```

The total cost equals 17. An optimal solution gives a total cost of 11 and it would look like this:

```
xxx             badness=3, cost=9
xx xx           badness=1, cost=1
xxxxx           badness=1, cost=1
```

Donald Knuth proves that such optimization can be efficiently accomplished using dynamic programming [http://en.wikipedia.org/wiki/Dynamic_programming] . According to Knuth, a paragraph can be modelled by an acyclic graph. The candidate lines represent the edges of the graph, and a numerical value—the demerit—is assigned to each edge. The paragraph selected is the one represented by the shortest path of the graph. Figure 9 shows the acyclic graph modeling the paragraph in the previous example.

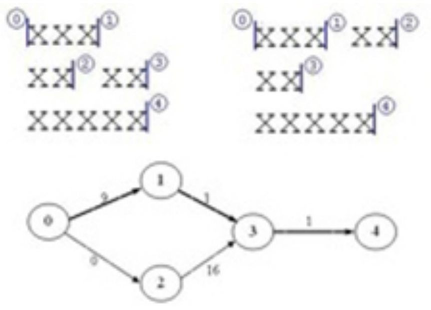


Figure 9:

Acyclic graph modeling the two versions of the same paragraph

A paragraph is a horizontal list that has to be broken into lines in an optimal way, depending on the run time as well as on the visual result. Of course, certain lines may be sacrificed to save the paragraph's justification quality. In the beginning, a paragraph is represented by a long list of elements, called *nodes*: character node, ligature node, discretionary node, math node, penalty node, glue node, or kerning node, which can be explicitly set by the user or implicitly provided by the font. On the basis of this list, the algorithm will generate another list of nodes called *feasible breakpoint nodes*.[5] [#N5] Each such node is characterized by three things:

- the position of this breakpoint in the horizontal list;
- the number of the line following this breakpoint;
- the fitness classification of the line ending at this breakpoint.

The strategy adopted by optimum-fit algorithm is:

1. Create an active node representing the beginning of the paragraph.
2. Run through the given horizontal list representing the paragraph.
3. For each legal breakpoint, run through the list of active nodes previously created.
4. Check if there is a way to achieve each feasible combination with the current breakpoint and create an active node in this place, if possible.
5. Assign a cost to the feasible line.
6. If two or more active nodes end a line at the same point, keep only the best one.
7. Choose the paragraph version with the optimal cost.

The algorithm creates the first active breakpoint node representing the beginning of the paragraph. The second active node is a feasible breakpoint depending on the first one; it can be a finite penalty node, a glue node, or a discretionary-hyphen node placed at an acceptable distance to make a potential line.

Acceptable distance

Let `L` be the inter-margin space and let `l` be the natural length of the material—boxes and glues—to put down in a line. Let `X` be the sum of stretching and `Y` the sum of shrinking of glues. If `l < L`, then the line needs to be stretched, and let `r = (L-l)/X,` a value needed to calculate the badness.

In practice, the line's badness `b` is given by `b=100 * r3`. If this value exceeds `10000`, the badness is treated as infinite. To save runtime, the algorithm tries first to make a paragraph without any hyphenation. The parameters *pretolerance* and *tolerance* are the limits on how bad a line can be before and after hyphenation, respectively. For simplicity, we will assume that hyphenation is never tried.[6] [#N6] The badness is compared to pretolerance. If b < pretolerance, the breakpoint is feasible. Next, the fitness classification will be calculated for the line that has just ended:

- *decent* if `0 ≤ b ≤ 12`
- *loose* if it has been stretched with `12 < b < 100`
- *very-loose*if it has been stretched with `b ≥ 100`.

As we advance in the horizontal list of nodes, sometimes `l` becomes greater than `L`, then the line needs to be shrunk, in which case the value `r = (l-L)/Y`. The fitness classification will then be calculated:

- *decent* if `0 ≤ b ≤ 12`

- *tight* if it has been shrunk with `b > 12`.

If `r > 1`, let `b = ∞ + 1`. It is important to distinguish between the situation where `b = ∞` and that where `b = ∞ + 1`. In the first case, the current active node should be still active; in the second case, the current active node should no longer be active.

Normally, if everything goes well, and badness is less than pretolerance, the current node is placed at an acceptable distance to make a potential line. For each potential line, a demerit is calculated, taking into account the badness and the different penalties, which include:

- hyphenation penalty;
- double hyphenation penalty;
- penalty for adjacent incompatible lines.

Based on the badness tolerated on each line, and because the system can expand or contract inter-word spaces, the algorithm considers various ways to achieve a feasible line from the current active node. The nodes created in this way will be linked in a list.

If we are too far from an active node, i.e., `l` is sufficiently large than `L`, that node is deactivated. The newly generated nodes will be used to identify other active nodes and so on, until the paragraph's end is reached. For formal reasons, the paragraph's end is considered to be a breakpoint. After all, it is the end of a line. Given the different nodes, we build an acyclic graph. The vertices of this graph are the different feasible breakpoints, and the edges are pairs of adjacent nodes making a feasible line. The problem is then to find the shortest path of the graph, choosing from among all the candidate paragraphs the version with the smallest possible value for the total demerit. Therefore, the paragraph is chosen after considering all its lines.

If the paragraph contains n breakpoints, the number of situations that are to be evaluated naively is 2^n. However, using the dynamic programming method, the complexity of the algorithm can be reduced to $O(n^2)$. Other simplifications can be evoked as well. For example, the system does not examine unlikely cases such as a break in the first word of the paragraph. This leads to an efficient algorithm with a running time almost of order n.

The following pseudo-code implements the optimum-fit algorithm:

```
active = [0]; nwords = length(paragraph)
for w in range(1,nwords)
    # Check the feasibility of breaking after the word w
    print "Recent word:",w
    for a in active
        line = paragraph[a:w+1]
        if w == paragraph[nwords-1]
            badness = 0 # last line will be set perfect
        else
            badness = compute_badness(line)
            print "...Line:" ,line, "; Badness:" ,badness,
        if badness > pretolerance
            active.deactivate(a)
            print "Active point:",a,"deactivated"
        else
            # Compute the cost of breaking after w
            update_demerit(a,w,badness)
            active.append(w)
```

Texteme and paragraph-breaking

The digital text representation model used today is the one proposed by the Unicode standard (Unicode Consortium 2006). It is based on the concepts of *character* and *glyph*. In the last few years, text processing tools have made use of characters first, while glyph issues are handled in the last stage, namely the rendering. Before being displayed, an e-document undergoes changes. In the beginning, a document is a string of structured characters. After processing, it becomes a string of glyphs arranged according to the rules of the typographic art.

Going from characters to glyphs and back can be a complicated process (Haralambous and Bella 2005a). One glyph can be associated with multiple characters and one character can be associated with multiple glyphs. Moreover, multiple glyphs can be associated with multiple characters in a different order.

Y. Haralambous proposes a new approach to the relationship between characters and glyphs: he merges them with a number of additional properties related to language presentation or other types of metadata into a single atomic text unit called *texteme* (Haralambous and Bella 2005b). A texteme is a set `{c, g, p1 = v1 ... pn = vn}`, where `c` is a character code, `g` a glyph index and a list of named properties p_i $(1 \leq i \leq n)$ taking value v_i. A simple texteme is a pair `(c, g)` and a pointer to an empty list of properties; text processing can be considered to be a process of enriching textemes with properties.

This concept of textemes, along with dynamic typography, points to a new approach to paragraph-breaking (Haralambous and G. Bella 2006). This approach starts with a paragraph as a linked list of textemes. The text processing enriches the textemes with alternative glyph properties whenever the current font provides an alternative form for such a glyph. The breaking algorithm takes into account the existence of such alternative forms to provide more flexibility and therefore more possibilities for breaking. This approach suggests thinking on a more theoretical level for solving problems that are admittedly fundamental, but still mostly practical.

Proposition for Arabic justification

Smart fonts

A modern typesetting system should be able to manipulate the so-called “smart font” formats, such as OpenType. In an Arabic rendering process, OpenType tables can carry out the following basic functions:

- supply the glyph corresponding to the pair (character, contextual form);
- supply grammatical ligatures (ex. Lam-Alef) and aesthetic ones;
- supply specific alternative forms for glyphs depending on context;
- supply kerning between single or ligatured glyphs;
- place short vowels and other diacritics on isolated glyphs or on ligatured components.

One could imagine some of these features being contextual, with or without backtrack and look-ahead. OpenType also provides what we can call “super-OpenType” features, including the “jalt” (justification alternative) feature (Microsoft Corporation 2009). The jalt table shows some glyph variants—wider or narrower letters that are intended to improve justification. WYSIWYG (what you see is what you get) software would probably show all the variants and offer the user the choice in an interactive manner. Or better, this feature can assist the paragraph builder engine to optimize the paragraph typographical gray during justification. For example, this feature in the Arabic Typesetting font provides us stretched alternative forms for fourteen glyphs (see table 1).

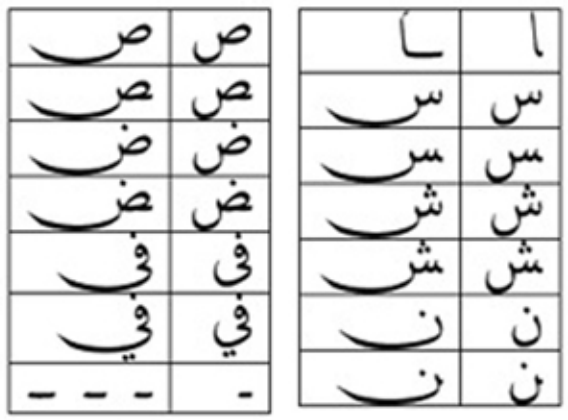


Table 1:

jalt table in Arabic Typesetting font

As we have seen above, Arabic writing provides various techniques from its handwriting traditions to perform text justification. These techniques are not based on the insertion of variable spaces between words or the horizontal scaling of glyphs, but in changing the shapes of letters themselves. That is how Arabic avoids unpleasant spaces and presents a balanced page.

A paragraph builder algorithm, either one based on a simple line-by-line justification or one based on a paragraph, can use other glyph variants—wider or narrower—to improve line justifications. These variants could be ligatures, allographs, or stretched glyphs with kashida. It's important not to break the strict rules governing the use of some allographs and to keep context in mind (Benaatia, Elyaakoubi, and Lazrek, 2006).

Such a text composition system, were it to be built, would allow us to distinguish between allographs for contextual use and glyph variants provided by the font for justification purposes. A "calt" (contextual alternative) table could specify the context where each substitution can occur. Thus, the contextual variants may be elements of the calt table while the variants used to improve justification may be elements of the jalt table.

Dynamic typography

Dynamic typesetting is a set of typesetting techniques where glyphs can be modified during the line break process. Gutenberg applied such techniques in the 42-line Bible. He was exploiting alternate forms, ligatures, and abbreviations to optimize justification on the line level.

Ligatures in the 42-line Bible were certainly a good way to achieve fine control over the widths of words. In Arabic script, ligatures are very numerous. Unlike Latin readers, Arabic readers are used to ligatures and our implementation is restricted to the use of simple substitutions rather than multiple substitutions or ligatures.

Our main motivation for designing a justification program is to add alternative forms to the paragraph-based line-breaking algorithm. That allows us to enhance the acyclic graph-modeling paragraph by introducing other breakpoint nodes so that the optimization of the paragraph will be improved. We recognize two breakpoint nodes that generate an acceptable distance for a potential line, and we build in a unique stretched alternative according to Arabic calligraphic rules (Benaatia, Elyaakoubi, and Lazrek, 2006).

For instance, suppose that a variable l contains the width produced by the group of glyphs and glues on the line, while L is the value where the line should be justified. We go through the horizontal list, and whenever a glyph node is encountered, we check if the jalt table provides another alternative. As long as we find no entries in the jalt, we proceed as in the ordinary optimum-fit algorithm. The value l is increased by the width of that glyph and the feasibility of the legal breakpoints is checked. This procedure consists of running through the active list to see what lines can be made from the active nodes to the current legal breakpoint node. An active node is generated for each feasible breakpoint and added to the active list.

If, however, the current node is a glyph with an alternative form in the jalt table, we calculate the difference between the two widths: the default glyph width and the alternative form width. The variable *backward* contains this difference. The widths of the two lines augment with a shift amount of *backward*. The badness is calculated twice, taking account of the *backward* produced by the alternative form.

We include two tests for the breakpoint feasibility: first using the default width, and second using the alternative form's width. If all goes well, and one of the two values $b_1$or b_2 is less than the *pretolerance*, the current legal breakpoint is a feasible one, based on the default glyph's use or the alternate form's use. We create an active node and we label this node to indicate from which glyph use it came. We indicate the position of the glyph in the node we have created. If that node comes from the two calculations, and both badness b_1 and b_2 satisfy the feasibility condition, both demerits d_1 and d_2 are calculated, and we keep only the one with the minimal demerit, since the second situation will never be chosen in the final paragraph breaks. This process allows us to omit a substantial number of feasible breakpoints from the rest of the process. If the two demerits are close in value, we keep the composition with the non-stretched glyph.

If two stretchable letters occur in the same line, we consider the two possibilities of stretching, giving us the opportunity to avoid situations such as two elongations in two consecutive lines. However, we do not consider combinations of the two possibilities and we follow the convention of Arabic text justification (Benaatia, Elyaakoubi, and Lazrek, 2006) that prohibits the use of two elongations in the same line.

If a stretched line justifies better than a non-stretched line, the program considers it in the context of the rest of the paragraph, and may resort to the non-stretched line to make the paragraph work better.

Authorized and prohibited elongations

We wanted to determine the contexts behind the use of kashida and allographs (Benaatia, Elyaakoubi, and Lazrek, 2006) and to include Arabic's graphical grammar. This study relies on an exploration in calligraphic compositions and historical references on the subject. Although our purpose was not to imitate handwriting, the best rules may be found in calligraphy. These rules take care not to replace the white "rivers" with black ones produced by successions of kashidas in consecutive lines. Some of these rules are mandatory, some are very common, and others are rare. Generally, when a calligrapher creates a visual work of art it is through intuition. We have tried to validate, formalize, and make some of that intuition explicit, but automating it is hard.

For instance, determining where to use kashida and allographs (Benaatia, Elyaakoubi, and Lazrek, 2006) can be based on graphic design—the number of kashidas and their degree of extensibility, their positions on the line, occurrences where stretching is allowed and so on—or semantic. For example, Vlad Atanasiu asserts that Ottoman calligraphers used the stretched Kaf, known as zinadi Kaf, to write the word *kufr* (disbelief) in order to stigmatize disbeliever persons (Atanasiu 2003).

Penalties are an efficient tool for communicating choices to the line-break process. Using penalties, a professional or a novice user can tell the system about his typographic and semantic preferences. In our model we associate penalties with each use of a stretched alternative form. These parameters specify the demerits added for lines. These penalties are independent from the line break; on the contrary, they depend on the use of the stretched alternative forms to achieve the line. Increasing the value of these parameters encourages the system to avoid prohibited elongations, even at the cost of other aesthetic considerations like avoiding too loose inter-word spacing. The user should assign a rather large value to these parameters to have an effect. The penalties themselves are of various types, explained below.

Position penalty

The historical references on the legitimate places for kashida differ in some details (Benaatia, Elyaakoubi, and Lazrek, 2006). In terms of justification, calligraphers use kashida at the end of a line because they can estimate the elongation only when they come near the limit of a line. Therefore, kashida is triggered by the distance from the end of line. Two elongations on two consecutive lines can only be seen as a defect. To avoid the “stairs” effect resulting from such superposition, a uniform typographical grid is useful.

In our model, for adjacent lines where two elongations are superposed, the system put a penalty on the second line. The superposition of three elongations on three adjacent lines evokes, of course, a higher penalty.

Occurrence penalty

By classifying words according to their number of letters, we can identify where stretching is allowed (Benaatia, Elyaakoubi, and Lazrek, 2006). Generally, it is strictly prohibited to stretch a word with two letters and usually kashida is not used in a word composed of three letters. Words with four letters are the most susceptible to stretching, and it is preferable to use kashida in the second letter.

We also incorporate rules concerning degrees of extensibility for each letter according to its context (Benaatia, Elyaakoubi, and Lazrek, 2006). For example, the elongation of letter Beh is authorized, but not approved, if it is followed by Alef, Jeem, Dal, Reh, and Lam; it is authorized and also approved if it is followed by Tah; and it is prohibited if it is followed by Ain, Seen, Feh, Qaf and Kaf. If elongation is authorized, the letter Beh can be stretched up to twelve diacritic points.

In our model, a pointer variable runs through the given horizontal list as we look for breakpoints. Another variable stays a step behind this pointer, and a second one stays two steps behind the pointer variable running through the horizontal list. These variables allow us to analyze the context of a glyph if it has an alternative form in the jalt. We add penalties where elongations are strictly prohibited. In the other cases, we should be parsimonious in the use of penalties in order to not compromise the typographical gray quality.

Semantic penalties

Contrary to the previous types of penalties, the semantic penalties depend on where the break occurs. Increasing the value of these penalties will encourage the system to avoid some semantically prohibited breaking, for example, between two words where a break is not recommended, such as within person’s name or inside a compound word.

Practical results

Basically, to implement our model, we needed two components: a font and a rendering technology. For the font, we used the Arabic Typesetting font distributed with the Microsoft Visual OpenType

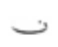
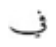
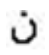
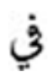
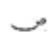
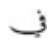
Layout Tool in the OpenType format. The Arabic Typesetting font contains the jalt table. We have used it here for illustration without altering it. Many shaped glyph forms, such as ligatures and alternative forms, have no Unicode encoding. These glyphs have GIDs (glyph identifiers) in the font, and applications can access these glyphs by "running" the layout features of these glyphs, particularly glyphs in the jalt table. This feature in the Arabic Typesetting font provides us some alternative stretched glyphs (see table 1 above), though the number of provided alternative forms is rather limited. [7] [#N7] We also omitted the use of the alternative form of final Alef in the first row of the table. We assume that a kashida is a forward extension of a glyph.

The second component we need is a rendering technology that implements a paragraph-breaking algorithm and takes advantage of what an OpenType font provides.

XƎTEX

is a typesetting system (Kew 2006) based on a merger of the T_EX system with Unicode and modern font technologies. It offers the optimum-fit algorithm and support for OpenType fonts.

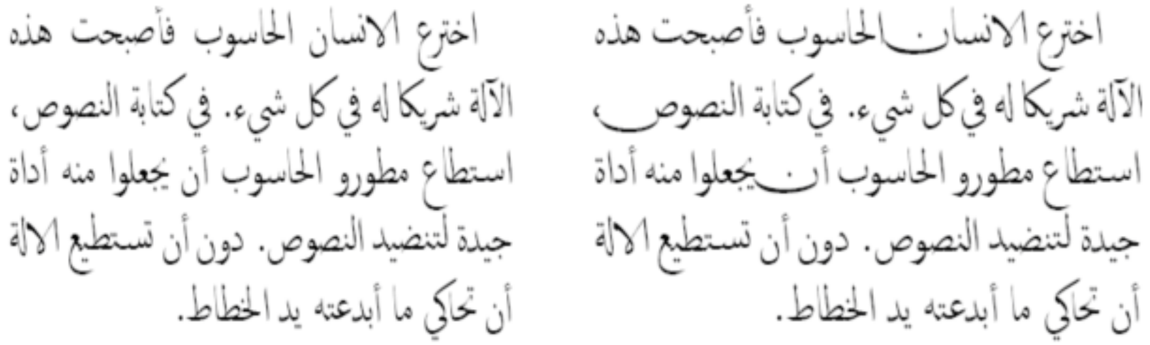
In Figure 10 [#articleID_001] , we illustrate the differences between a paragraph processed with the optimum-fit algorithm and the same paragraph processed with our improved algorithm. In this example, note the algorithm's choice of glyph 1 instead of the default (glyph 2) in the first and third line, and glyph 3 instead of glyph 4 in the second line. In the second line, the optimum-fit algorithm would have chosen glyph 5 instead of the first glyph 6, resulting in a position penalty since this substitution shows up below an elongation in the first line. It would also have chosen glyph 7 instead of the second glyph 6, introducing another penalty position as it shows up above an elongation in the third line of the paragraph.

	
Glyph 1	Glyph 5
	
Glyph 2	Glyph 6
	

Glyph 3 Glyph 7

ص

Glyph 4



اخترع الانسان الحاسوب فأصبحت هذه
الآلة شريكا له في كل شيء. في كتابة النصوص،
استطاع مطورو الحاسوب أن يجعلوا منه أداة
جيدة لتنضيد النصوص. دون أن تستطيع الالة
أن تحاكي ما أبدعته يد الخطاط.

اخترع الانسان الحاسوب فأصبحت هذه
الآلة شريكا له في كل شيء. في كتابة النصوص،
استطاع مطورو الحاسوب أن يجعلوا منه أداة
جيدة لتنضيد النصوص. دون أن تستطيع الالة
أن تحاكي ما أبدعته يد الخطاط.

Figure 10:

Left, regular typesetting; right, typesetting with our improved optimum-fit

Conclusion

In this paper, we have explored how Arabic text justification differs from other writing systems, and presented a significant model that improves the typographical gray quality obtained with the optimum-fit algorithm, an encouraging step toward providing a complete model that observes the strong rules of Arabic text justification.

One issue yet to be addressed is the use of multiple substitutions of glyphs, such as ligatures and abbreviations, to improve Arabic text justification. To do so, one must be able to enable or disable them dynamically, according to the requirements of each line of text and within the constraints of legibility.

The Web issues require further study beyond the scope of this paper. These days, designers can specify margins, padding, and letter spacing in a Web page in a better way, thanks to cascading style sheets. Therefore, on the Web, the size, the resolution, and even the layout of page can vary with the browser and the system used to view the site. Thus, the development of an efficient justification system for the Web becomes a delicate problem.

Mohamed Elyaakoubi is a Ph.D. student in the department of Computer Science at Cadi Ayyad University. He is a member of the Multilingual Scientific E-Document Processing Team. His current main research interest is multilingual typography, especially publishing Arabic e-documents that follow the strong rules of the Arabic calligraphy.

Email: m.elyaakoubi@ucam.ac.ma

Azzeddine Lazrek is full Professor in Computer Science at Cadi Ayyad University in Marrakesh. He holds a Ph.D. in Computer Science from Lorraine Polytechnic National Institute in France, awarded in 1988, and a State Doctorate Morocco awarded in 2002.

Prof. Lazrek specializes in communication through multilingual multimedia e-documents. His areas of interest include multimedia information processing and its applications, particularly, to electronic publishing, digital typography, Arabic processing, and history of sciences. He is in charge of the Information Systems and Communication Networks Research Team and the Multilingual Scientific E-Document Processing Research Group. He is an Invited Expert at W3C. He leads a multilingual e-document composition project with some international organizations. He contributes to scientific journals and is a member of several national and international scientific associations.

Email: lazrek@ucam.ac.ma

Acknowledgments

We would like to express our appreciation to Judith Axler Turner for her editorial corrections and valuable suggestions for improving this final version. We also recognize the contributions of anonymous reviewers whose rigorous critiques, comments, and suggestions made this paper possible. Thanks are also due to Khalid Sami for his priceless comments and corrections to an earlier draft of the manuscript.

References

Atanasiu, Vlad. 2003. “Allographic Biometrics and Behavior Synthesis.” EuroT$_E$X 2003 Proceedings. *TUGboat* 24, no. 3: 328–33.

Bayar, Abdelouahad, and Khalid Sami. 2009. “How a Font Can Respect Basic Rules of Arabic Calligraphy.” *International Arab Journal of e-Technology* 1, no. 1.

Benaatia, Mohamed Jamal Eddine, Mohamed Elyaakoubi, and Azzedine Lazrek. 2006. “Arabic Text Justification.” *TUGboat* 27, no. 2: 137–46.

Berry, Daniel M. 1999. “Stretching Letter and Slanted-baseline Formatting for Arabic, Hebrew, and Persian with ditroff/ffortid and Dynamic PostScript Fonts.” *Software: Practice & Experience* 29, no. 15: 1417–57. [doi:10.1002/(SICI)1097-024X(19991225)29:15<1417::AID-SPE282>3.0.CO;2-F [http://dx.doi.org/10.1002/(SICI)1097-024X(19991225)29:15<1417::AID-SPE282>3.0.CO;2-F]]

Châtry-Komarek, Marie. *Tailor-Made Textbooks, A Practical Guide for the Authors of Textbooks for Primary Schools in Developing Countries*. Oxford: CODE Europe, 1996.

Comber, Tim. 1994. “The Importance of Text Width and White Space for Online Documentation.” BAppSc (Hons) Thesis, Southern Cross University.

Fine, Jonathan. 2000. “Line Breaking and Page Breaking.” *TUGboat* 21, no. 3: 210–21.

Fredkin, Edward. 1960. “Trie memory.” *Commun. of the ACM* 3, no. 9: 490–500. [doi:10.1145/367390.367400 [http://dx.doi.org/10.1145/367390.367400]]

Haralambous, Yannis. 1995. “Tour du monde des ligatures.” *Cahiers GUTenberg* no. 22: 87-99.

Haralambous, Yannis, and Gábor Bella. 2005a. “Injecting Information into Atomic Units of Text.” *Proceedings of the 2005 ACM Symposium on Document Engineering*, Bristol, U.K., pp. 134–42.

Haralambous, Yannis, and Gábor Bella. 2005b. “Omega Becomes a Texteme Processor.” EuroT$_E$X 2005 Proceedings (Pont-à-Mousson, France), pp. 99–110.

Haralambous, Yannis, and Gábor Bella. 2006. “Fontes intelligentes, textèmes et typographie dynamique.” *Document numérique* 9, nos. 3–4: 167–216.

Hochuli, Jost, and Robin Kinross. 1996. *Designing Books: Practice and Theory*. London: Hyphen Press.

Hssini, Mohamed, Azzeddine Lazrek, and Mohamed Jamal Eddine Benatia. 2008. "Diacritical Signs in Arabic E-Document" (in Arabic). CSPA'08 [http://www.phillips-publishing.com/cspa08/] , The 4th International Conference on Computer Science Practice in Arabic, Doha, Qatar, April 1-4.

Karow, Peter. 1997. "Le programme *hz*: micro-typographie pour photocomposition de haut niveau." *Cahiers GUTenberg* no. 27: 34–70.

Kew, Jonathan. 2006. "Unicode and Multilingual Typesetting with $\mathrm{X_ET_EX}$." *TUGboat* 27, no. 2: 228–29.

Knuth, Donald E. 1986. *TEX: The Program, Computers and Typesetting*, vol. B. Reading, Mass.: Addison-Wesley.

Knuth, Donald E., and Michael F. Plass. 1981. "Breaking Paragraphs into Lines." *Software: Practice & Experience* 11, no. 11: 1119–84. [doi:10.1002/spe.4380111102 [http://dx.doi.org/10.1002/spe.4380111102]]

Lazrek, Azzeddine. 2003. "*CurExt,* Typesetting Variable-Sized Curved Symbols." $\mathrm{EuroT_EX}$ 2003 Proceedings. *TUGboat* 24, no. 3: 323–27.

Liang, Franklin Mark. 1983. "Word Hy-phen-a-tion by Comput-er." Ph.D. thesis, Stanford University.

Microsoft Corporation. 2009. OpenType Specification, version 1.6. http://www.microsoft.com/typography/otspec/ [http://www.microsoft.com/typography/otspec/]

National Institute of Advanced Industrial Science and Technology (AIST). 2009. *libotf — Library for handling OpenType fonts*. http://www.m17n.org/libotf/ [http://www.m17n.org/libotf/]

Peck, Wendy. 2003. *Great Web Typography*. New York: Wiley Publishing, Inc.

Sherif, Ameer M., and Hossam A. H. Fahmy. 2008. "Parameterized Arabic Font Development for AlQalam." *TUGboat* 29, no. 1: 79–88.

Thành, Hàn Thê. 1999. "Améliorer la typographie de $\mathrm{T_EX}$." *Cahiers GUTenberg* 32 (actes du congrès GUT'99, Lyon, May): 21–28.

The Unicode Consortium. 2006. *The Unicode Standard*, version 5.0. Upper Saddle River, N.J.: Addison-Wesley.

Wild, Adolf. 1995. "La typographie de la *Bible* de Gutenberg." *Cahiers GUTenberg* no. 22: 5–15.

Zachrisson, Bror. 1965. *Studies in the Legibility of Printed Text*. Uppsala: Almqvist & Wiksell.

Zapf, Hermann. 1993. "About Micro-Typography and the *hz*-Program." *Electronic Publishing* 6, no. 3: 283–88

Notes

1. We use "glyph" in the sense of Unicode, which defines the glyph as a specific shape that a character (letter, number or symbol) can have when rendered or displayed. [#N1-ptr1]

2. Such places are called points of *legal break*. [#N2-ptr1]
3. URW company (Unternehmensberatung Rubow Weber—from the founders' names). [#N3-ptr1]
4. The imaginary line upon which the letters of a font rest. [#N4-ptr1]
5. Real candidates for breaking, both legal and probable. [#N5-ptr1]
6. Since the 10th century, hyphenation in Arabic writing has been strictly prohibited. [#N6-ptr1]
7. We are developing an Arabic font observing strong Arabic calligraphy rules in the OpenType format. [#N7-ptr1]