Arabic text justification using LuaLaTeX and the DigitalKhatt OpenType variable font

Amine Anane

Abstract

Arabic script is a cursive script where the shape and width of letters are not fixed, but vary depending on the context and justification needs. A typesetter must consider these dynamic properties of letters to achieve high-quality text comparable to Arabic calligraphy.

This article presents a proof-of-concept implementation of Arabic text justification, by varying letter shapes and widths, as a first step towards such high-quality Arabic typesetting. It uses LuaLaTeX and the DigitalKhatt OpenType variable font produced from a METAFONT-based dynamic font.

1 Introduction: Justification in Arabic

Due to the cursive nature of Arabic script, where the letters are connected, a letter can have several distinct forms and width depending on its position in the word and the letters that surround it. For example, Figure 1 shows fourteen possible forms of the letter ب (beh), among which are extensible forms.

Therefore, unlike the Latin script where justification is mainly based on the distribution of whitespace, Arabic calligraphy takes advantage of this dynamic nature of letters to justify text using three main techniques, as explained below: kashida extension, wider form substitution, and ligature composition and decomposition.

1.1 Kashida extension

The curvilinear stroke that connects letters is called kashida or tatweel. The width of the kashida is variable and can be stretched or compressed to justify text. Table 1 shows kashida extension examples.

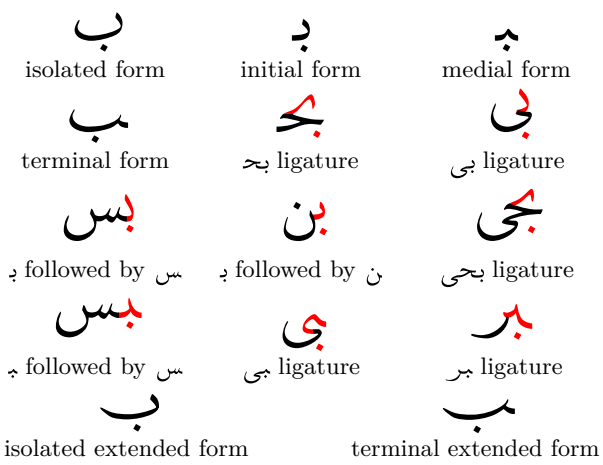


Figure 1: letter ب (beh) shapes

Table 1: Kashida extension

Min width	Natural width	Max width
حه	حه	حـــه
صا	صا	صـــا
حد	حد	حـــد
سو	سو	ســـو
با	با	بـــا

1.2 Wider form substitution

Several Arabic letters have extended forms whose width can vary continuously within an interval. To stretch a line, a wider form is substituted. Table 2 shows some examples of wider forms.

Table 2: Wider form substitution

Original letter	Wider form	
	Min width	Max width
ب	ب	بـــ
س	س	ســـ
ن	ن	نـــ
ف	ف	فـــ
ك	كـ	كـــ

1.3 Ligature composition/decomposition

Arabic script makes extensive use of ligatures [2]. To stretch a line, an optional ligature can be decomposed to its constituent letters. Conversely, a less common ligature can be composed to shrink a text line. Table 3 shows some examples of ligature decomposition.

xnotdoi.org/10.47397/tb/0-0/tb132anane-variable

Table 3: Ligatures

Ligature	Constituent letters	
	Natural width	Max width
بـح	بـح	بـــح
لـح	لـح	لـــح
هـم	هـم	هـــم

To implement justification using the three techniques above, a dynamic font having variable glyph width and shape can be designed. By applying the justification rules provided by the font, the justification process will determine the width and shape of the glyphs to stretch or shrink a line of text. A few previous works [1, 2, 3, 12, 14] have been interested in Arabic justification using dynamic fonts. Unfortunately, these works still have significant challenges to overcome and they do not appear to be any further along in years.

On the other hand, most standard font formats, such as Apple Advanced Typography (AAT), Graphite and OpenType, provide mechanisms to control justification of text lines. However, Arabic justification has not benefited much from these solutions. For instance, the idea of using a variable font to stretch glyphs to do justification dates from TrueType GX, currently specified in AAT by the means of the `just` table. Unfortunately, this feature cannot be used since it is not supported by the Apple CoreText layout engine.

So, the approach adopted in this project is to start from the widely used OpenType standard and extend it to support Arabic justification using dynamic fonts, while drawing inspiration from existing solutions.

This article presents a proof-of-concept implementation of Arabic text justification by applying the three techniques above, using LuaLaTeX and the DigitalKhatt OpenType variable font. Section 2 introduces METAFONT and the VisualMETAFONT editor, used to design the DigitalKhatt font. Section 3 presents the OpenType layout engine, the justification algorithm and rules that have been developed. Section 4 shows an overview of OpenType variable fonts and explains how such a font is generated from the METAFONT. Section 5 gives the results obtained using LuaLaTeX and compares them with a handwritten Quranic text. Section 6 concludes with future work needed to further improve the results.

2 METAFONT

To design the glyph of an extensible letter, a function must be defined. This function produces shapes representing the extensible letter at different widths. At runtime, the justification process will determine the appropriate function arguments to justify text lines by stretching or shrinking some glyphs.

To design such parametric glyphs, the METAFONT [11] language[1] is used, which was specifically designed to deal with parametric fonts. Here are some of the advantages of METAFONT.

- It supports macros to extend the language with new syntax and operations.
- It supports deducing smooth cubic Bézier curves from high-level constructs such as direction, tension, curl, without having to specify each control point of the cubic curve.
- It supports a declarative style using linear equations.

For example, from the following METAFONT code

```
draw (0,0) .. (10,0) .. cycle
```

the following curve is generated: ◯. METAFONT deduces the control points in such a way to have a smooth curve as close as possible to a circle. This default behavior can be changed by giving the tension or direction at some points. The following code

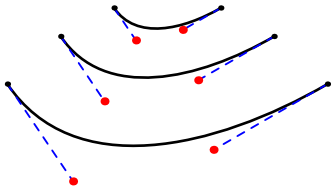
```
draw (0,0) .. tension 1.5 ..
    (10,0) .. tension 1.5 .. cycle;
```

applies some tension to the end points to obtain an ellipse-like shape: ⬭. This feature of automatically calculating the control points can be useful to define an extensible kashida. As an example to illustrate this, the following code

```
draw (0,0){dir -56}
  .. {dir 30} (50 + extension,0);
```

produces the following shapes by setting 0, 50 and 100 to the variable `extension`. METAFONT has automatically shifted the control points linearly with respect to the variable `extension`, allowing it to keep the same curve shape.

[1] Specifically, the METAPOST program [7, 8, 9] was used, an altered version of METAFONT which produces vector graphic commands instead of raster images. The term 'METAFONT' is kept since it is used for font and meta-font design (i.e., functions generating functions).



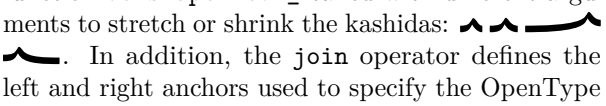
In addition to the kashida having variable width, it can have several shapes depending on the connecting letters. To simplify the design of the font, a step-by-step approach has been adopted. As a first step, a generic stretchable kashida has been used to join most of the connecting letters. So a functional parametric font can be obtained as quickly as possible, from which justification experimentation can be started. Subsequently a gradual refinement of the kashidas will be undertaken, as mentioned in section 6.

To design such a generic kashida a new operator `join` is added. The following METAFONT code uses this new operator to define the glyph `behshape.medi`.

```
defchar(behshape.medi,-1,-1,-1,-1);
%%beginparams
z1 = (219.986,47.4628);
z3 = (134.906,31.769);
%%beginverbatim
leftExtRatio := 8;
rightExtRatio := 10;
righttatweel_const := (120,-y1);
z2 = z1 + (penwidth,0) rotated 70;
z4 = z3 + (penwidth,0) rotated 83;
%%beginpaths
fill (181.269,99.7718) {dir -116.055} .. z3
    join z4 .. tension 0.932615 and 2.98102 ..
    (182.631,170.321) .. tension 1.7264 and
    1.23542 .. {right}(195.366,180.062) ..
    tension 1.7264 and 1 .. z2 join z1 .. {curl
    1} cycle;
%%endpaths
enddefchar;
```

The `join` operator can accept arguments to have more control over the form of the kashida. For example, the `leftExtRatio` and `rightExtRatio` variables above influence the curvature of the left and right kashidas. The `defchar` command generates the "glyph function"

$$\langle glyph\_name \rangle \texttt{\_(leftExt,rightExt)} \qquad (1)$$

which produces the glyph shape as a function of the argument values provided. Here is the result of the function `behshape.medi_` called with different arguments to stretch or shrink the kashidas:  . In addition, the `join` operator defines the left and right anchors used to specify the OpenType cursive attachment connections between glyphs.

Another advantage of METAFONT is the possibility of using elliptical and polygonal pens to describe curves. However, this feature was not used, in order to easily generate from the METAFONT font an OpenType version which can be used elsewhere. So all the glyph shapes are described using only cubic curve outlines.

On the other hand, a major drawback of METAFONT is that a font designer has to write code to design a glyph. To validate that the glyph is correct, it must be visualized by executing the code and generating the glyph image with some labeling points to help debug the code and fine-tune the result. This process repeats until getting the desired result.

This fine-tuning process becomes more tedious if there are several parameters, since it is important to see how the glyph behaves by changing the parameter values. Also, if the font design is based on tracing of handwritten letters, as it is the case in this project, the task becomes even slower.

So, a visual graphic editor was developed to overcome this limitation, called VisualMETAFONT. This editor is based on Qt, a C++ software development framework. VisualMETAFONT uses the `mplib` library [9], a project supported by the LuaTEX team in order to turn the METAPOST interpreter into a system library that can be used by other applications. It is also based on HarfBuzz, a text-shaping engine supporting OpenType, AAT, and Graphite used in many applications and devices such as LuaLaTeX, Android, Chrome, Firefox, XeTeX and Qt.

Figure 2 shows the VisualMETAFONT window for glyph design. It is composed of three panes. The left pane is the METAFONT code of a single glyph. Each glyph starts with a `beginchar` or `defchar` command which provides its name, Unicode code point and glyph dimensions, if applicable. The code for a glyph can have five sections. The first section is a path to an image file that will be imported, to be traced around. The second section defines the parameters that will be controlled graphically. The third section is free METAFONT code to add other variables and define the relationships between them. The fourth section defines the paths that will be manipulated graphically. The fifth and last section is free METAFONT code. Therefore, there is no limit on what can be used in the METAFONT code. On the other hand, there are limits to what can be managed graphically, although features can be added to the GUI as needed.

The outline resulting from the execution of the METAFONT code is shown in the middle pane. The points that can be manipulated graphically are shown with different colors depending on their functions

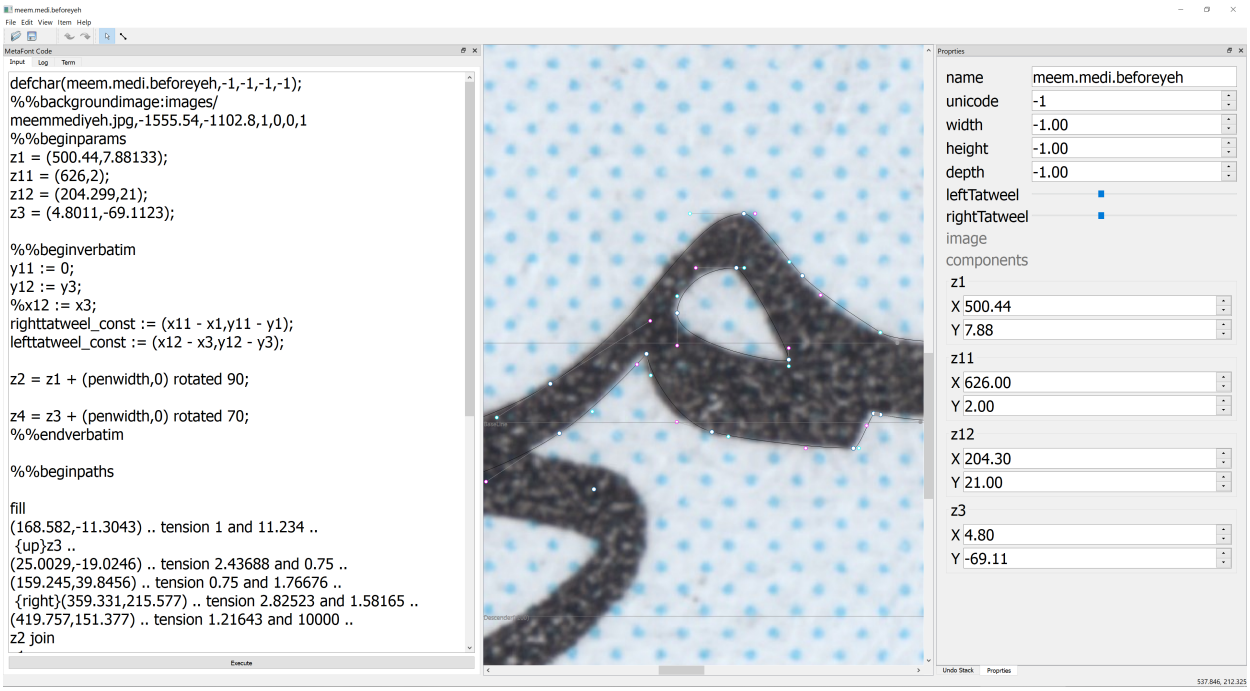


Figure 2: VisualMETAFONT glyph window

(i.e., inline point, left and right control points, parameter of type point). Each change in the position of one of these points generates new METAFONT code which is reflected immediately in the visual pane.[2]

The right pane displays the glyph parameters which can also be controlled graphically using slider and spin widgets, especially for numeric parameters without a graphic representation in the middle pane. For instance, by sliding the widgets for `leftTatweel` and `rightTatweel`, the user can view how the glyph width and shape behave for different values.

Once the dynamic glyphs are designed, a font designer needs to define the OpenType and justification rules that govern how a Unicode-encoded text will be presented using these glyphs. The next section presents these rules and the algorithms behind them.

3 Justification

This section starts by introducing the OpenType layout engine and its mechanism of lookups and rules. Next, the justification process is presented; it inherits the same notions of lookups and rules.

The OpenType layout engine takes as input an ordered list of lookups given a language and a set of features. Each lookup contains an ordered list of rules having the same type of action, which is either glyph substitution or glyph positioning. A substitution action replaces one sequence of glyphs by another. It is used for different purposes such as composition, decomposition, ligature, and alternates. A positioning action alters the advance width and offset position of a glyph. It is used for kerning, cursive attachment, mark to base position or mark to mark positioning. An OpenType rule can be considered as a simple form of pattern matching with the following general form:

$$\textit{backtrack input lookahead} \rightarrow \textit{action}$$

If the input, backtrack and lookahead sequences match the glyph string at the current glyph then the layout engine applies the action of the rule to the input sequence and advances the current glyph to the glyph after the input sequence. The layout engine does the substitution lookups first, then the positioning lookups, as shown in Algorithm 1.

The same principle of lookups and rules is used for justification. Indeed, by applying ligature compositions and decomposition, glyph alternates, kerning, and cursive attachment it is possible to shrink or extend a line until desired width is achieved. This technique of justification has been used in the Oriental TEX Project [6].

The current implementation uses the same idea. Two sets of steps can be defined: one used to shrink text lines and one used to stretch text lines. Each step contains a set of lookups that will be applied in

[2] Due to the speed of METAFONT and contemporary machines, the visual design is very smooth and no slowness or lagging is experienced.

Algorithm 1: OpenType layout algorithm

```
input  : Unicode text
output : List of glyph id, advance widths,
         offset positions
glyphRun ← map Unicode text to glyph id;
foreach stage of {substitution, positioning} do
  foreach lookup of stage's lookups do
    foreach currentGlyph in glyphRun do
      foreach rule of lookup do
        if rule matches at currentGlyph then
          apply rule's action to input sequence;
          currentGlyph ← glyph after input seq;
          break;
        end
      end
    end
  end
end
```

the order given until reaching the desired line width. Listing 1 shows an example justification table, which will be discussed in section 5.

Listing 1: Justification table

```
table(just) {
  stretch {
    lookup expa.ligadecomp;
    lookup expa.kafalternate;
    lookup expa.alternates;
    lookup expa.seensad;
    lookup expa.taamar_haa_dal;
    lookup expa.alef_tah_lam_kaf;
    lookup expa.behshape_yeh_reh;
    lookup expa.waw_ain_qaf_fa;
    lookup expa.others_fina;
  }
  shrink {
    lookup shr1.ligatures;
    lookup expa.shrinkspace;
    lookup shr1.kaf;
    lookup expa.shrink;
    lookup shr1.kern;
    lookup shr1.dalcursive;
  } };
```

To take into account the dynamic aspect of the glyphs, a new lookup format is added. The idea is to assign for each expandable glyph two attributes defining its ability to stretch and to shrink, as with TEX glue. (As future work, it would be interesting to study the possibility of including these attributes in breaking of paragraphs into lines by TEX's line-breaking algorithm.) Listing 2 shows an example of

Listing 2: Justification lookup

```
lookup expa.taamar_haa_dal {
  feature sch1;
  lookupflag IgnoreMarks;
  lookup expa.haa.l1 {
    sub behshape.init expfactors 0 5 0 0;
    sub behshape.medi expfactors 0 5 0 0;
    ...
    sub heh.fina expfactors 0 0 0 2;
    sub dal.fina expfactors 0 0 0 2;
  } expa.haa.l1;
  ...
  sub @haslefttatweel'lookup expa.haa.l1
     [heh.fina dal.fina]'lookup expa.haa.l1;
} expa.taamar_haa_dal;
```

this new format of justification lookup. The class `@haslefttatweel` in this example contains all the glyphs having an extensible kashida. If one of these glyphs is followed by a `heh.fina` or `dal.fina` then the justification attributes defined by the lookup `expa.haa.l1` are used.

So, depending on the context, the lookup specifies which glyph will be substituted by another, if any, in addition to the maximum desired stretch or shrink for the left and right kashidas. For example, Listing 2 specifies that `behshape.init` and `behshape.medi` can stretch by an argument value of `leftExt` that cannot exceed 5 and `heh.fina` and `dal.fina` can stretch by an argument value of `rightExt` that cannot exceed 2.

The justification proceeds in two phases. The first phase constitutes the execution of the lookup, as given by Algorithm 1. The result of this execution is the set of glyphs to be substituted, including their extension attributes. The second phase calculates the stretch/shrink values and applies them to the glyph string, after substituting the affected glyphs.

Even though there are different justification techniques, as given in section 1, from the point of view of the justification algorithm all these techniques constitute substitutions. Indeed, the kashida technique is a special case of justification by alternates, which is a substitution of one glyph by another — only in this case it may or may not be the same glyph. The same is true for ligature composition and decomposition, which is a substitution of one sequence of glyphs with another, and thus represents a more general case than the two previous techniques. The algorithm takes into account this general case.

Here is the problem stated more formally. As the result of the first phase, we will have m sequences

S_k to be substituted, where each sequence contains one or more glyphs. Let Δw_i be the amount that will be added to, or subtracted from, the width of the line, if the glyph i is substituted, and let e_i its maximum stretchability or shrinkability. Let $\Delta W_k = \sum_{i \in S_k} \Delta w_i$. Let Δd the amount by which we want to stretch or shrink a text line. So the problem of the second phase is to find the set of sequences R and the extensibility ratio r such as

$$\sum_{k \in R} (\Delta W_k + \sum_{i \in S_k} e_i * r) \leq \Delta d$$

If we want to find the solution which maximizes the above sum in order to minimize the gap Δd of the line, the problem becomes NP-hard. Indeed, if we ignore the term $\sum_{i \in S_k} e_i * r$ from the inequality above, the problem becomes an optimization problem of the subset sum problem, which is NP-hard.

We can think of some heuristics to solve this problem, or we can use an exact algorithm, when there are only a few substitutions to apply. In addition to minimizing the gap Δd, other criteria can be considered. For example, we may prefer to distribute the substitutions throughout the line rather than being concentrated at the beginning or at the end of the line. A deeper study of the rules of justification will dictate the way forward.

Currently, the algorithm implemented iterates over the sequences and chooses a sequence if its gap ΔW_k is less than the remaining line gap Δd, otherwise it skips to the next sequence. We can consider ordering the sequences such that the glyphs to be justified are at the left or the right of the line or distributed over the entire line. Algorithm 2 gives the steps in more detail.

For Algorithm 2 to work, it is necessary that the width difference Δw_i for each glyph i be linear in its arguments `leftExt` and `rightExt`, in the intervals between their current values and their maximum values specified by the justification rule.

Algorithm 2: Justification lookup algorithm

input : m sequences S_k and Δd
output : a set of sequences R and a ratio r
totExpansion $\leftarrow 0$;
$R \longleftarrow \emptyset$;
for $k \leftarrow 1$ **to** m **do**
  if $\Delta W_k \leq \Delta d$ **then**
    $\Delta d = \Delta d - \Delta W_k$;
    totExpansion $\leftarrow$ totExpansion $+ \sum_{i \in S_k} e_i$;
    $R \longleftarrow R + \{S_k\}$;
  end
end
if totExpansion $> \Delta d$ **then**
  $r \leftarrow \Delta d$/totExpansion;
  $\Delta d \leftarrow 0$;
else
  $r \leftarrow 1$;
  $\Delta d \leftarrow \Delta d -$ totExpansion;
end

4 OpenType variable fonts

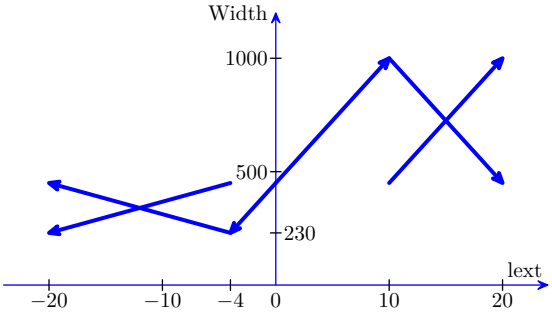
Variable fonts originated in Apple's TrueType GX and was adapted by OpenType in 2016. As variable fonts are used more and more, it is interesting to show that the justification technique already implemented with METAFONT also works for variable fonts.

Initially, the goal of using variable fonts was to allow continuous variations along design axis such as weight, width, slant, italic and optical size. An axis has a minimum, maximum and default values. For a given axis value, the layout engine use interpolation over regions of the variation space to calculate variable quantities such as glyph outline points, anchor position, and glyph advance width.

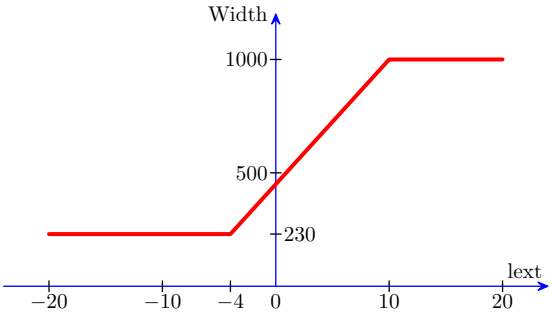
The same idea can be applied to glyph extension. In our case, two extension axes are defined, `lext` and `rext` corresponding to the `leftExt` and `rightExt` parameters. Each axis has a default value 0. The minimum and maximum values of the axes are chosen arbitrarily, and each glyph can decide how the axes' values are mapped to the argument values of the corresponding parameters.

To simplify the explanation below, we will assume that the minimum value of an axis is the minimum possible argument value of its corresponding parameter, and likewise for the maximum. A mapping that only requires two regions by glyph quantity is to return the minimum value of a quantity when the minimum value of the axes are given, and likewise for the maximum. However, this mapping can make the width difference Δw_i of the previous section non-linear in the interval containing 0 (i.e., the functions are discontinuous at 0) and therefore the justification rules should be adapted to this situation, and probably also Algorithm 2.

So, to get linear mapping functions all along the axis, the minimum values are returned when the minimum argument value of the glyph is given, and likewise for the maximum. For any axis value less than the minimum argument value, its minimum values are returned, and conversely for the maximum. This second mapping which preserves linearity was used to generate the OpenType variable font. So each glyph is represented with a maximum of six regions for each axis, three for the negative axis and three for the positive axis.



(a) Variation regions



(b) Resulting width function

Figure 3: Variation example

As an example, suppose that the minimum value for the `lext` axis is −20 and the maximum value is 20. Further suppose we have a glyph with a natural width of 450. Its maximum width is 1000, obtained when the argument value is 10, and the minimum width is 230, obtained when the argument value is −4. Figure 3a shows the 6 regions that should be defined for this glyph. Since the results of the regions in the same segment add up, we get the resulting width function in Figure 3b which becomes constant when it reaches its maximum or minimum value.

At this point, the following question may arise: What is the advantage of using METAFONT if variable fonts can meet the need of a dynamic font to apply justification? As highlighted in section 2, METAFONT remains an excellent tool to design such a variable font, thanks to its power and flexibility. In addition, if METAFONT could be used during the layout process then its full power can be harnessed. Nevertheless, designing a dynamic glyph using interpolation can be easier than trying to adjust the direction and tension of the curve points, especially when there are multiple points to define. Fortunately, METAFONT can design a glyph as an interpolation function between two shapes. Also, it is often not possible to find the glyph with the desired maximum width. In this case, extrapolation can be used to deduce the maximum shape.

5 Experimental results

A prototype implementation was presented at the TUG 2019 meeting. The implementation is a patched version of HarfBuzz and `mplib` and accepts as input a METAFONT font. That implementation was compiled to WebAssembly and a demonstration was published at `www.digitalkhatt.org`.

This section presents the result of a prototype implementation based on LuaLaTeX which uses as input the DigitalKhatt OpenType variable font presented in the previous section. All the examples presented here are compiled with this patched version of LuaLaTeX.

LuaLaTeX supports OpenType fonts through the `luaotfload` package, which is an adaptation of ConTeXt modules. This package has three OpenType processing modes: `base`, `node` and `harf`. The `base` mode supports only OpenType features that can be mapped to traditional TeX ligature and kerning mechanisms, and the `node` mode uses the OpenType layout engine implemented in ConTeXt.

The `harf` mode [10] uses the HarfBuzz library linked in LuaHBTeX. Since HarfBuzz is already part of LuaHBTeX, the integration of the justification algorithm becomes much easier. Indeed, the justification algorithm is patched in HarfBuzz and linked with LuaTeX to generate a patched version of LuaHBTeX. The Lua interface to the HarfBuzz engine is also modified to include the new functionality.

Unlike `luaotfload`'s `node` mode, `harf` mode does not support OpenType variable fonts. So the `node` modules supporting variable fonts are adapted to `harf` mode. In the `node` mode, variable glyphs are treated as LuaTeX virtual characters with special commands. During PDF generation, these commands are inserted each time the character is referenced, potentially leading to large file sizes. To remedy this, the virtual characters are converted to real characters of Type 3 fonts, created dynamically during PDF generation. This conversion also has the advantage of treating glyphs like characters, thus enabling PDF text functionality.

As a case study, the justification technique proposed by Microsoft to extend CSS2 and implemented in Internet Explorer (IE) is adapted here using only justification rules and therefore shows that the justification algorithm is general enough to support multiple justification scenarios.[3] Here is a description of the technique, as specified in [13].

1. Find the priority of the connecting opportunities in each word.

[3] This technique has been abandoned and to date there is no new CSS proposal for Arabic justification.

2. Add expansion at the highest priority connection opportunity.
3. If more than one connection opportunity has the same highest value, use the opportunity closest to the end of the word.

Table 4 defines the priority for connection opportunities and where expansion occurs. Since the algorithm deals with automatic justification, priority 1 for manual insertion of the kashida is ignored.

The IE implementation requires one kashida per word. To simplify the justification rules, this constraint is reduced to one kashida per module (i.e., cluster of connecting letters). IE also uses a CSS attribute `kashida-space` which defines the amount of justification given to kashidas in ratio to spaces. In our case, the amount of stretch is specified by the justification rules. When there is no more stretching possible, spaces will be used.

The CSS specification deals only with kashidas and does not mention the other justification techniques such as glyph alternates and ligature decomposition. In this case study, these techniques are added according to the priorities specified in the justification table of Listing 1. Optional ligature decomposition has the highest priority, followed by Kaf alternate, then by the other alternates.

Afterwards, kashida justification rules are defined according to the priorities presented in Table 4. These rules consider that the glyphs with a given priority participate in the justification process only after glyphs with higher priorities have exhausted their maximum stretchability or shrinkability. The following example compares the IE result (top) to the LuaLaTeX result (bottom), using the Times New Roman font after applying kashida justification. The line is stretched about 1.4 times its natural width.

إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ

إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ

IE justifies text by inserting flat kashidas unlike the curvilinear stretching of the current solution, which applies kashidas by substituting a wider glyph. Figure 4 shows how the text line stretches and shrinks, changing the line width by 5pt each time. Line 6 is set to the natural width of the text which gives a non-justified text. So the first five lines correspond to the shrinking case. Lines 5 and 4 are shrunk by decreasing the space between words due to the lookup `expa.shrinkspace`. Lines 3 and 2 shrink kashidas due to the lookup `expa.shrink`. The result of the shrink is similar to that obtained by using the pdf/LuaTeX font expansion feature. In this case, however, the expansion factor is a continuous value whose limit can depend on the context and the priority level. Line 1 uses in addition the lookup `shr1.dalcursive` to decrease the space before ـنَ in the second word.

1 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
2 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
3 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
4 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
5 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
6 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
7 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
8 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
9 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
10 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
11 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
12 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
13 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
14 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
15 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
16 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
17 إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ
إِنَّ ٱلَّذِينَ كَفَرُواْ سَوَآءٌ عَلَيْهِمْ ءَأَنذَرْتَهُمْ أَمْ لَمْ تُنذِرْهُمْ

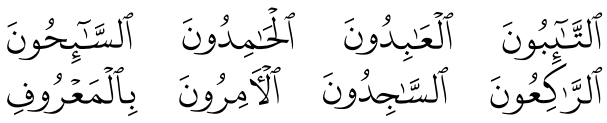
Figure 4: Continuous expansion

Line 7 shows two types of justifications. The first is the decomposition of the ligature ـهُمْ in the last word, and the second is the kashida stretching of the letter seen in سَوَآءٌ , having priority 2 in Table 4. Line 8 uses the alternate wider glyph ـنَّ of the letter noon ـنَّ of the first word. However, the previous letter سَ is no longer stretched. This is because, in line 7, there is not enough space to accommodate the wider ـنَّ, while in line 8 there is enough space, and since the alternates have priority over kashidas, the wider ـنَّ is chosen. In line 9, the wider alternate of letter kaf ـكَ is used. Since it has priority over ـنَّ, the wider noon ـنَّ has shrunk a little. At line 10, the second alternate ـنَّ of the second word is substituted. From line 10 to line 13, the widths of ـنَّ and ـنَّ increase continuously until they reach their limits; at this moment, the stretching of the letter سَ comes into play again. At line 14 starts the stretching of نذَ and نذَ, of priority 3, until reaching their limits at line 18 where begins the stretching of kashidas of priority 7, which continues to the last line 19.

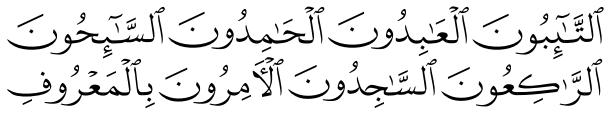
Table 4: Priority for kashida opportunities in CSS/IE proposal

Pri	Glyph	Condition	Kashida location
1	User-inserted Kashida	The user entered a Kashida in a position.	After the user inserted kashida
2	Seen, Sad	Connecting to the next character (initial or medial form).	After the character.
3	Taa Marbutah, Haa, Dal	Connecting to previous character.	Before the final form of these characters.
4	Alef, Tah, Lam, Kaf, Gaf	Connecting to previous character.	Before the final form of these characters.
5	Ra, Ya, Alef Maqsurah	Connected to medial Baa	Before preceding medial Baa
6	Waw, Ain, Qaf, Fa	Connecting to previous character.	Before the final form of these characters.
7	Other connecting characters	Connecting to previous character.	Before the final form of these characters.

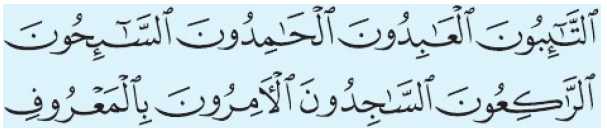
Figure 5 shows an example which compares the justification result of the current solution with the same handwritten text from which the font was designed. In the second line the wider alternates are not distributed evenly. For instance, the final noon ن of the second word has not been substituted by a wider noon ﻧ, unlike the other two letters noons. Instead, there is some stretching before the final Reh ر of the third and fourth words which has not been considered among the justification cases of Table 4.



(a) justified text using interword space



(b) justified text using letter stretching



(c) handwritten text

Figure 5: Comparison with handwritten text

Figure 6 shows another example illustrating the difference between the handwritten text and LuaLaTeX.

So far, all the examples presented are typeset in TeX's restricted horizontal mode using `\hbox`. Figure 7a shows an example of justified text using the current technique after TeX breaks paragraph into lines. Figure 7b shows the same text not justified. Some kashidas remain to be improved; however, this justification technique based on the continuous extension of letters conforms to Arabic calligraphy principles, allowing for better quality of Arabic text.

6 Future work

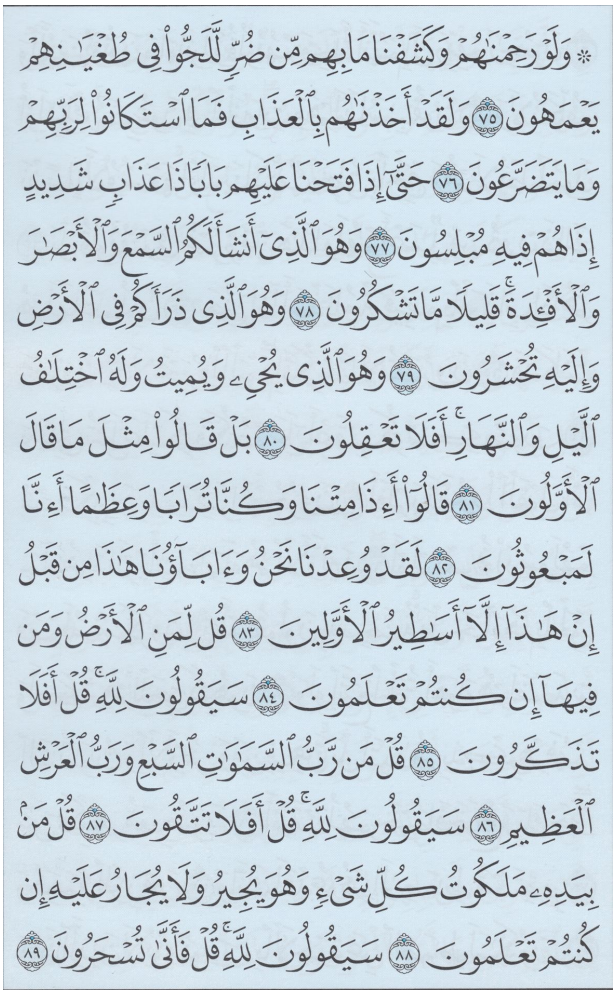
This article has presented a proof-of-concept implementation of Arabic justification based on letter extension techniques used in Arabic calligraphy, using LuaLaTeX and the DigitalKhatt variable font, which has achieved very promising results. This section presents some future work in the short and medium-term to continue the journey towards a high quality Arabic typesetter.

As a first step it is important to do more analysis and experiments of the different calligraphic rules used for Arabic justification to answer some important questions, such as: Which justification approach is preferable over another and in what context? If there are many justification choices with same preference, how should the space be distributed among them? Some works [2, 4, 6] have studied these rules and it would be interesting to implement them and compare them with the handwritten text.

Secondly, the DigitalKhatt font needs to be improved, especially at the kashida level. For the moment, each connecting glyph starts and/or finishes at

*وَلَوْ رَحِمْنَٰهُمْ وَكَشَفْنَا مَا بِهِم مِّن ضُرٍّ لَّلَجُّوا۟ فِى طُغْيَٰنِهِمْ
يَعْمَهُونَ ﴿٧٥﴾ وَلَقَدْ أَخَذْنَٰهُم بِٱلْعَذَابِ فَمَا ٱسْتَكَانُوا۟ لِرَبِّهِمْ
وَمَا يَتَضَرَّعُونَ ﴿٧٦﴾ حَتَّىٰٓ إِذَا فَتَحْنَا عَلَيْهِم بَابًا ذَا عَذَابٍ شَدِيدٍ
إِذَا هُمْ فِيهِ مُبْلِسُونَ ﴿٧٧﴾ وَهُوَ ٱلَّذِىٓ أَنشَأَ لَكُمُ ٱلسَّمْعَ وَٱلْأَبْصَٰرَ
وَٱلْأَفْـِٔدَةَ ۚ قَلِيلًا مَّا تَشْكُرُونَ ﴿٧٨﴾ وَهُوَ ٱلَّذِى ذَرَأَكُمْ فِى ٱلْأَرْضِ
وَإِلَيْهِ تُحْشَرُونَ ﴿٧٩﴾ وَهُوَ ٱلَّذِى يُحْىِۦ وَيُمِيتُ وَلَهُ ٱخْتِلَٰفُ
ٱلَّيْلِ وَٱلنَّهَارِ ۚ أَفَلَا تَعْقِلُونَ ﴿٨٠﴾ بَلْ قَالُوا۟ مِثْلَ مَا قَالَ
ٱلْأَوَّلُونَ ﴿٨١﴾ قَالُوٓا۟ أَءِذَا مِتْنَا وَكُنَّا تُرَابًا وَعِظَٰمًا أَءِنَّا
لَمَبْعُوثُونَ ﴿٨٢﴾ لَقَدْ وُعِدْنَا نَحْنُ وَءَابَآؤُنَا هَٰذَا مِن قَبْلُ
إِنْ هَٰذَآ إِلَّآ أَسَٰطِيرُ ٱلْأَوَّلِينَ ﴿٨٣﴾ قُل لِّمَنِ ٱلْأَرْضُ وَمَن
فِيهَآ إِن كُنتُمْ تَعْلَمُونَ ﴿٨٤﴾ سَيَقُولُونَ لِلَّهِ ۚ قُلْ أَفَلَا
تَذَكَّرُونَ ﴿٨٥﴾ قُلْ مَن رَّبُّ ٱلسَّمَٰوَٰتِ ٱلسَّبْعِ وَرَبُّ ٱلْعَرْشِ
ٱلْعَظِيمِ ﴿٨٦﴾ سَيَقُولُونَ لِلَّهِ ۚ قُلْ أَفَلَا تَتَّقُونَ ﴿٨٧﴾ قُلْ مَنۢ
بِيَدِهِۦ مَلَكُوتُ كُلِّ شَىْءٍ وَهُوَ يُجِيرُ وَلَا يُجَارُ عَلَيْهِ إِن
كُنتُمْ تَعْلَمُونَ ﴿٨٨﴾ سَيَقُولُونَ لِلَّهِ ۚ قُلْ فَأَنَّىٰ تُسْحَرُونَ ﴿٨٩﴾

(a) LuaLaTeX



(b) Page 347 of Madina Mushaf

Figure 6: Difference between LuaLaTeX and handwritten text

the minimum of the kashida. It is up to the justification rule to specify how much width the left curve of the kashida has and how much width the right curve has. A better approach would be to design connecting letters as a single stroke containing the whole kashida, and then the justification rule specifies the width of the whole kashida. Using METAFONT, the kashida will be split at specific points to produce each glyph part.

Another important work to consider is to embed code within the font in order to support custom rules. For instance, the width of some marks depends on the width of other glyphs, and custom rules are needed to express this. In a recent presentation [5], the author of HarfBuzz has proposed using WebAssembly to embed code within a font, since current technology allows it, which would bring great flexibility to the font developer. For instance, if this happens, it will be possible to embed the justification algorithm based on METAFONT within the font, thus allowing to take full advantage of METAFONT and to use it in any platform supporting this technology. So, as future work, WebAssembly will be considered to implement custom lookups and actions instead of using a specific scripting language. Also, to deal with complex Arabic calligraphic rules, adding custom lookups supporting the full power of regular expression and finite state machines could be necessary.

When typesetting the Quran by respecting the handwritten Mushaf, page by page and line by line, the difficulty will be more in terms of line shrinking than line stretching. Indeed, the calligrapher can visually optimize the space between letters and words so that some letters can appear above others and some marks can change their positions horizontally and vertically to fill the space, while avoiding some marks or letters being too close. Implementing this using

(a) justified text using letter stretching (b) justified text using interword space

Figure 7: Text justification after TEX's line-breaking algorithm

rules will be very difficult due to the large number of possible cases. An algorithm which analyzes the available space to move marks and word modules and prevent collisions would be very useful.

Finally, it is interesting to study how to extend TEX's line-breaking algorithm in order to consider glyph stretching and shrinking properties before breaking paragraph to lines.

Acknowledgments

I would like to thank Barbara Beeton and Karl Berry for improving this article with their English and technical editing.

References

[1] A. Bayar, K. Sami. How a font can respect basic rules of Arabic calligraphy. *Intl. Arab. J. e-Technol.* 1(1):1–18, 2009. cs.uwaterloo.ca/~dberry/HTML.documentation/KeshidePapers/HowFontCanRespectArabicCalligraphy.pdf

[2] M.J.E. Benatia, M. Elyaakoubi, A. Lazrek. Arabic text justification. *TUGboat* 27(2):137–146, 2006. tug.org/TUGboat/tb27-2/tb87benatia.pdf

[3] D.M. Berry. Stretching letter and slanted-baseline formatting for Arabic, Hebrew, and Persian with ditroff/ffortid and dynamic PostScript fonts. *Softw. Pract. Exp.* 29:1417–1457, 1999. doi.org/10.1002/(SICI)1097-024X(19991225)29:15%3C1417::AID-SPE282%3E3.0.CO;2-F

[4] M. Elyaakoubi, A. Lazrek. Justify just or just justify. *J. Elect. Pub.* 13(1), Winter 2010. doi.org/10.3998/3336451.0013.105

[5] B. Esfahbod. Better-engineered font formats. www.youtube.com/watch?v=fG1QEcl3yks&ab_channel=BehdadEsfahbod

[6] H. Hagen, I.S. Hamid. Oriental TEX: optimizing paragraphs. *MAPS* 45:128–154, 2012. www.ntg.nl/maps/45/12.pdf

[7] J.D. Hobby. A METAFONT-like system with PostScript output. *TUGboat* 10(4):505–512, Dec. 1989. tug.org/TUGboat/tb10-4/tb26hobby.pdf

[8] J.D. Hobby. *A User's Manual for MetaPost*, 1994. www.tug.org/docs/metapost/mpman.pdf

[9] T. Hoekwater, L. Scarso. *MPlib API documentation, version 2.00*, 2018. mirror.ctan.org/systems/doc/metapost/mplibapi.pdf

[10] K. Hosny. Bringing world scripts to LuaTEX: The HarfBuzz experiment. *TUGboat* 40(1):38–43, 2019. tug.org/TUGboat/tb40-1/tb124hosny-harfbuzz.pdf

[11] D.E. Knuth. *The METAFONTbook*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[12] S. Mansour, H. Fahmy. Experiences with Arabic font development. *TUGboat* 33(3):295–298, 2012. tug.org/TUGboat/tb33-3/tb105mansour.pdf

[13] P. Nelson. Justifying text using cascading style sheets (CSS) in Internet Explorer 5.5/6.0. web.archive.org/web/20030813215144/http://www.microsoft.com/middleeast/Arabicdev/IE6/KBase.asp

[14] A. Sherif, H. Fahmy. Meta-designing parameterized Arabic fonts for AlQalam. *TUGboat* 29(3):435–443, 2008. tug.org/TUGboat/tb29-3/tb93sherif.pdf

◇ Amine Anane
ananeamine (at) gmail dot com
https://github.com/DigitalKhatt