Egyptian Computer Science Journal Vol. 37 No. 5 September 2013 ISSN-1110-2586

An Algorithm to Justify Arabic Text

[1]Aqil M. Azmi, [2]Abeer Alsaiari

[1]Department of Computer Science, King Saud University, Riyadh, Saudi Arabia
[2]Department of Computer Science, Taibah University, Madina, Saudi Arabia
aqil@ksu.edu.sa , abeer_alsaiari@yahoo.com

Abstract

The majority of the tools geared for e-documents composition have been tailored for the Roman based script's needs. The localization of these tools to Arabic alphabet based writings is not an easy task, due to the great typographical and structural characteristic differences, besides the Arabic letters are context sensitive. With recent advances in Arabic typography, it still is far from the aesthetics developed by centuries old practice of Arabic calligraphy. In this paper we tackle one aspect of Arabic calligraphy, and that is text justification. A common scheme to justify the Arabic text is through kashida, which is elongation of the connecting line between the letters. Though common, this scheme does not follow the proper Arabic writing aesthetics. In this paper we develop a powerful algorithm for the justification of Arabic text that does not rely on kashida alone, as is the current practice.

Keywords: *Arabic typography, font, kashida, typesetting, text justification.*

1. Introduction

The Arabic language is native to roughly three hundred million people. The Arabic script is one of the most used in the world, not only by Arabs but also by the Muslim world as it is the script used to write the Qur'an, the holy book of Muslims. Moreover, the Arabic script is used, in various slightly extended versions, to write many major languages, e.g. Farsi, Urdu, Pashto, Sindhi ... etc. It is constituted in its basic form by 28 letters including 3 long vowels. In addition there are short vowels, a total combination of 13.

The process of typesetting languages using the Arabic script is more challenging and complex than typesetting using the Roman script because of the requirement for special needs and strict rules. The aesthetics of Arabic calligraphy gives the writing different situations according to the analysis of the context and the calligraphers rules. But most of the current typesetting software do not take into account the aesthetic aspect of the Arabic calligraphy.

When justifying a Latin based text, the typesetting software rely on: (1) hyphenation; and (2) insertion of spaces between words. On the other hand, the Arabic based typesetting software stretches the words vertically using kashida (**كشيدة**). Such a solution is unacceptable from the Arabic calligraphic principles where certain letter can be stretched while others can be compressed, see Fig. 1.

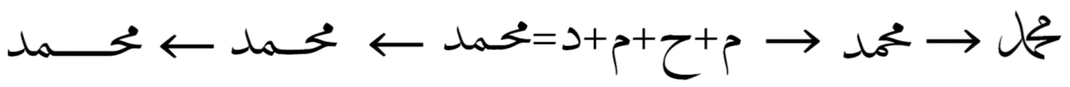


Figure 1. The name Muhammad at different degrees of stretching (left) or compressing (right).

During the first and second century of Hijra, there were no mandatory rules for Arabic script. The drawing of letters did not follow any particular style of specific rules. There are many examples of Mushafs scribed in the first and second century Hijra with a word split into two lines, Fig. 2.

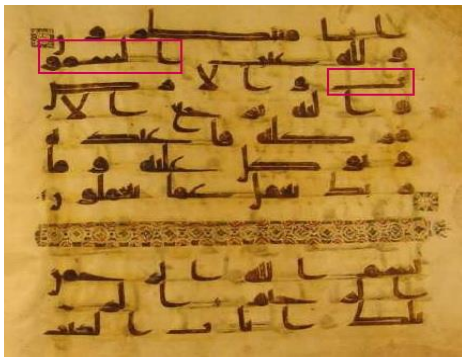


Figure 2. Mushaf Uthman at al-Mashhad al-Husaini Mosque, Cairo.
This page has two words split into two lines, we marked one of them (السموت) in red rectangle.

Copying the Qur'an by skilled scribes and calligraphers led to the emergence of Arabic calligraphy art and many styles for writing were developed, e.g. Naskh, Riqa'a, Thuluth. Master calligraphers devised basic rules for Arabic calligraphy, and these rules were evolved by later masters. One of the earliest master calligrapher is Ibn Muqla (d. 328 Hijra). He is known to be behind six scripts including the three scripts mentioned above [11]. It is believed that the Arabic script reached its height through the hands of master calligrapher, Yaqut Al-Musta'simi (d. 696 Hijra). And as thus, the Arabic writing became distinguishable with a group of features and rules that we would take into account in order to produce an acceptable text from the point of view of Arabic calligrapher [1], [6].

This set of strict rules have been maintained ever since and we would like to keep them intact into the digital age. This compels us to develop new rules for Arabic typography through the development of appropriate algorithms specifically to justify the Arabic text.

In this paper we aim to formalize the Arabic calligraphy rules through powerful algorithms to achieve a high quality output that conforms to some of the calligraphers' rules regarding text justification.

2. Related Work

As we mentioned, Arabic calligraphy has some very strict rules. For example, hyphenation is not allowed in Arabic as is the case in the Latin languages [6]. Also, the Arabic script is based on the cursive style of writing. This cursivity implies four different forms for the same letter according to its position in the word. The general four basic forms of a letter are: at the start of a word, at the middle, at end, and isolated. In terms of tools designed for text processing, the cursivity brings new constraints to Arabic typography. The

Arabic script does not enjoy the same luxury that Latin script has when it comes to automated typesetting on computers [6]. The development of Arabic documents processing tools needs to formalize the Arabic handwriting rules. For example, the Arabic calligraphy is extensible by itself [3]. So, justification of the text using a kashida must consider their localization rules. Regrettably most of the typesetting systems use a straight line kashida due to the limitation of using dynamic fonts. According to proper Arabic calligraphy, we need to dynamically generate a Bezier curve and use it as kashida.

Knuth and MacKay [9] were the first to present a working solution for including right-to-left text (for Arabic and Hebrew) in the TeX family. Their proposed TeX-XeT system is an extension of TeX. Within the TeX extensions, both Omega and ArabTeX have been used for Arabic and have met some of the basic requirements to varying degrees.

Haralambous [8] presented an infrastructure for typesetting in the Arabic script. This infrastructure is based on four tools: the concept of texteme, OpenType fonts, Omega 2 modules and an extended version of TeX’s line-breaking graph; which texteme is an atomic unit of text consists of character data (such as Unicode position) and another related data [3][8]. The tool resulted from this infrastructure is called the extended TeX graph. When applying the functionality of this infrastructure at each step of Arabic text processing, this will create an Arabic textemes containing all the information accumulated through the Arabic text processing steps, as well as the initial information: Unicode characters, contextual forms, etc. At the end, this is lead to an ideal infrastructure for typesetting in the Arabic script.

AlQalam [7] grew as a modifications to ArabTeX. Hence, it inherits ArabTeX’s good features. It is intended for typesetting the Qur'an, other traditional texts, and any publication in the language using the Arabic script. In [10], a new approach for developing Arabic font was presented. In order to achieve an output quality close to that of Arabic calligraphers, they tried to model the pen nib and the way it is used to draw curves as closely to the ideal as possible using METAFONT. Parameterized fonts were also introduced for a more flexible and dynamic combination of glyphs, to be used in forming ligatures and in drawing whole words as single entities.

Ditroff/ffortid is a system for formatting bi-directional text in Arabic, Hebrew and Persian [5]. The system is able to format mixed left-to-right and right-to- left texts using fonts with isolated letters or with connecting letters and only connection stretching, achieved by repeating fixed-length baseline fillers.

3. Arabic Writing Characteristics

In terms of Arabic type design, the typographer must take into account a number of characteristics and rules of Arabic script. A good awareness of these characteristics leads to professional design of Arabic type. We will go over some of them briefly.

3.1 Direction of writing

Arabic is a unidirectional script in which the writing spreads out from right-to-left. Nowadays, Arabic mathematical documents adopt Latin alphabetic symbols which has led some to believe that Arabic writing is bidirectional because of the current mixing of Arabic symbols with Latin based expressions [6].

3.2 Cursivity

The Latin based writing is based on the use of independent characters. In Arabic, only the cursive style is allowed. This cursivity implies four different forms for the same letter, for example the letter *ba*, according to its position in the word: initial (بـ), middle (ـبـ), final (ـب) and isolated: (ب) [2][6].

3.3 Ligatures

Arabic script is extremely rich in ligatures due to the cursive nature of writing. Some ligatures are mandatory while others are optional and exist only for aesthetic reasons, legibility or justification [2]. The ligatures can appear in various degrees, see Fig. 3.

لمحة ← لمحة ← لمحة

محمد ← محمد ← محمد

Figure 3. Ligatures in Arabic can appear in various degrees.

3.4 Diacritic dot

Diacritic dots are a measurement unit marked by the feather of the used calligraphy pen [4][6]. The semantic role of diacritic dots is that certain letters are characterized by the presence, number and positions of these dots [4]. For example, the basic glyph ٮ gives several letters according to the number of diacritic dots which appear above or below: ب، ت and ث. It is also used by calligraphers as a measurement unit to regularize the dimensions and the metrics of glyphs, Fig. 4. [4][6].

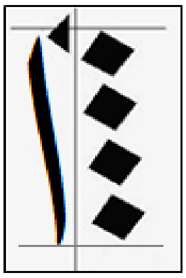


Figure 4. Using dots as a measurement unit.

3.5 Diacritic signs

Diacritic signs (or short vowels) are markings added above or below the letters to aid in proper pronunciation of the purely consonantal text. The diacritic signs take different heights, not only with respect to basic glyphs but also according to other contextual elements. The Arabic letter can be compared to a magnet for the diacritic mark [1][6].

3.6 Allograph

These are different graphical form a letter can have while keeping its place, i.e. initial, middle, final and isolated. Its form is dependent on the neighboring letters and the presence of kashida. For example, the initial form of the letter *ba* can take three different allograph shapes (Fig. 5) according to its left neighboring letter [4].

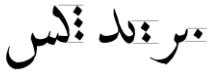


Figure 5. Three different allographic shapes of the initial letter *ba*. The diacritic dots are used to indicate the height.

3.7 Kashida

Kashida is a connection between Arabic letters and it is not a separate character but a stretch of the previous letter; used for various purposes: emphasis, legibility, aesthetic and justification [5]. As with ligatures, the kashida comes in various degrees, Fig. 6. The stretching of a letter is not haphazard, but rather follows a set of rules which defines the priorities and degrees of which a letter can be extended. This set of rules are stored in what is known as kashida matrix (Table 1) [3][4]. According to the calligraphic rules the maximum a letter can be stretched is twelve times its actual length.

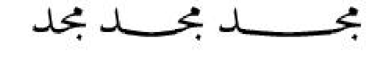


Figure 6. Kashida extension at various degrees. Note the curvature of the kashida.

Table 1. The kashida matrix. Blank means elongations are not allowed, '+' highly allowed, and '-' allowed but discouraged. Letters inside [ ] means all the characters with the same skeleton.

	Preceding letter																
Current letter	–	أ	[ب]	[ج]	[د]	[ر]	[س]	[ص]	[ط]	[ع]	ف	ق	ك	ل	م	ن	ه	و
[ب]	5	1		-1	-1	-1			+1					-1	1	1	1	
[ج]		-1		-1	-1	-1			+1	-1			-1	-1	-1	-1	-1	-1
[ط][ص][س]	3	1	2	-1	-1	1	2	2	1	-1	-1	-1	2	2	-1	2	-1	-1
[ع]		-1		-1	-1	-1			+1	-1				-1	-1	-1	-1	
ف ق	1	1		-1	-1	-1			+1	-1				-1	-1			-1
ك	3	-1		-1	-1	1		-1					-1	-1	-1	-1	-1	-1
ل	3			-1	-1					-1					-1	-1	-1	
م		-1		-1	-1	1			1	-1			-1	-1	-1	-1	-1	-1
ه					-1	-1	-1										-1	

4. The Proposed Scheme

The letters in Arabic can either be compressed or stretched with each of the actions can be done in various degrees. Our proposed text justification scheme is a two level process. First, we substitute the composed ligatures with alternative forms of less/more width according to the available space. Second, we repeatedly apply the kashida according to the kashida matrix till the line reaches the appropriate width. Most of the available Arabic fonts support limited ligatures and that means they are poorly suitable for our proposed algorithm. So before implementing the algorithm we need to prepare an appropriate font. Designing a font from scratch is time consuming process and requires a professional design artist. We decided to pick an existing font and edit it by adding some new ligatures. In this section we will start with our font development and then move to the algorithm.

4.1 Font enhancement

To avoid a timely process of developing font from scratch we decided to pick a suitable font and add to it some important ligatures. For this we picked the earliest version of Arabic Typesetting Font (ATF), an OpenType font that was designed mainly by Mamoun Sakkal for Microsoft Corporation. The font is built on the basis of Naskh script, and differs slightly from the official version which was distributed with MS Office 2007. This prelease version of the font comes with all the OpenType tables, which is not the case with the official release.

The font contains over 2,100 glyphs, including contextual alternates, ligatures, and language specific forms. Great care and consistency has been applied in glyphs used for many other languages, e.g. Farsi, Urdu ... that uses a variant of basic Arabic script. Table 2 shows the most common ligatures in ATF as well as the new ligatures which we added. In addition we added kashidas of different elongation into the ATF font.

Table 2. List of common and new ligatures in the ATF font. The red colored ligatures are missing in the original ATF and were added in our adaption. Also the ligatures under Variations are from our adaption.

Isolated	Initial	Middle	Final	Variations
كا	كا	ـكا	ـكا	
بم	بمـ	ـبمـ	ـبم	
كم	كمـ	ـكمـ	ـكم	
كل	كلـ	ـكلـ	ـكل	
لا	لا	ـلا	ـلا	
لم	لمـ	ـلمـ	ـلم	
سح	سحـ	ـسحـ	ـسح	
صح	صحـ	ـصحـ	ـصح	
سم	سمـ	ـسمـ	ـسم	
صم	صمـ	ـصمـ	ـصم	
بح	بحـ	-	-	
حج حح	حجـ حخـ حخـ	-	-	
عج عح عخ	عجـ عحـ عخـ	-	-	
فخ	فخـ	-	-	
لح	لحـ	-	-	
مح	محـ	-	-	
هج هح هخ	هجـ هحـ هخـ	-	-	
حم	حمـ	-	-	
عم	عمـ	-	-	

Continued Table 2. List of common and new ligatures in the ATF font. The red colored ligatures are missing in the original ATF and were added in our adaption. Also the ligatures under Variations are from our adaption.

Isolated	Initial	Middle	Final	Variations
فم	فمـ	-	-	
مم	ممـ	-	-	
هم	همـ	-	-	
بي	-	-	ـبي	
سي	-	-	ـسي	
صي	-	-	ـصي	
في	-	-	ـفي	
كي	-	-	ـكي	
لي	-	-	ـلي	
مي	-	-	ـمي	
هي	-	-	ـهي	
سر	-	-	ـسر	
صر	-	-	ـصر	
له	-	-	ـله	
لد	-	-	ـلد	
لك	-	-	ـلك	
-	بمـ	ـبمـ	-	
-	سهـ	ـسهـ	-	
-	صهـ	ـصهـ	-	
-	-	ـما	ـما	
-	-	ـبر	ـبر	
-	-	ـبن	ـبن	
لحم	لحمـ	-	-	
لمح	لمحـ	-	-	
لحح	لححـ	-	-	
-	محمـ	-	-	
-	لهـ	-	-	
-		-	-	
لهم	-	-	-	

Following the introduction of new ligatures into ATF, we need to tell the font when to compose these ligatures. This facility is available through the OpenType feature glyph substitution. Though FontLab provides support to OpenType features, VOLT (Visual OpenType Layout Tool) (http://www.microsoft.com/typography/volt.mspx) allows it visually. For example, if we have the glyph *uniFD8A* (ﶊ) and is followed by the glyph *uniFEAA* ﺪ) then these two glyphs are substituted by a single glyph *glyph2940* (ﶊﺪ). By this way we determine when to compose each one of these ligatures. The font is now ready to be used once it is compiled.

4.2 The proposed algorithm

We consider a document in a word processor as simply a set of paragraphs, with each paragraph ending with a newline character. To justify a line, the algorithm computes its badness value which is the value resulting from subtracting the width of the current line from the specific total width (which is the maximum width of a line). The badness value is always positive because the program will not allow reading a line larger than the specific total width. After computing the badness value, the algorithm starts its justifying process until the badness value becomes zero, actually when badness falls below a certain threshold we treat it as zero. The process of justification is a two level process: substitution level and kashida level.

4.2.1 Substitution level

After the badness is computed, the algorithm starts looking for any character that has an alternative glyph of a wider width. It only considers the alternative glyphs that have a width equal or less than the badness value. From all the alternative glyphs that have been found, it picks the one with the largest width. This ensures we minimize the search for alternative glyphs that can fit the badness value. After determining the best alternative glyph, the algorithm re-computes the badness value. The process of looking for new glyph and replacement is continued till the badness value becomes zero or if there are no more appropriate alternative glyphs that can fit the badness value. In the latter case the algorithm proceeds to the next level that is working with kashida.

4.2.2 Kashida level

In this stage, the algorithm determines the best positions to insert the kashida. The goal is to fill in the gaps and reduce the badness value to zero.

We set priorities as to which word(s) should have the kashida. The shorter the word, the higher priority it has to have a kashida (Table 3). Four letter words have the highest priority, followed by those having five letters and so on. The priority for which words should be stretched first follows the calligraphic rules, where kashida appears more frequently in four letter words and less in five letter words and even lesser in six letter words. Kashida is not recommended for two of three letter words with some exceptions in cases such as: (سر) and (صر) for two letter words and (بسم) for three letter word.

Table 3. Priority levels of words for holding the kashida

Length of words	Priority
Four letters	4
Five letters	3
Six letters	2
Seven letters	1

Once we have determined which words should have the kashida. Next we do need to decide which characters(s) within the word should be stretched. Here we need to consult the kashida matrix. The entries with '+' sign have the highest priority (which we set as priority 3), this is followed by entries with no sign. The entries with '−' sign have the least priority, i.e. priority 1.

The process of determining the best position for kashida starts by looking first for the word with the highest priority for having the kashida. Then, looking for the character in this word with the highest priority to be stretched by kashida according to the kashida matrix. In case there is no appropriate character in the current word, we move to the next word with the highest priority and so on. When the best position for kashida has been determined according to previous scenario, the algorithm will take the badness value as a parameter and will insert the kashida in the determined position. The elongation of the kashida is determined by the badness value, but in no case it can exceed 12 times the character's length. After the kashida is inserted, the badness value is recomputed. We continue looking for next place to insert kashida until badness value becomes zero. This scenario of justification is repeated from the top of the document till its end.

5. Results and Discussion

The algorithm cannot work by itself but it has to be integrated with a text editor software. The text editor will do all of the typical task and our algorithm will be called to justify the text before being rendered for screen output. For our implementation we used 'neatpad' (http://www.catch22.net/tuts/neatpad), an open source text editor that supports Arabic.

Any software that renders text will be composed of three basic components. The first component intercepts the stream of character codes from the keyboard sending them to the font engine. The second component is the Uniscribe based font engine which translates the characters codes into a stream of glyphs via the font's tables. These are returned back to the application for rendering on the screen. The third and the last component is the font typeface (Figure 7).

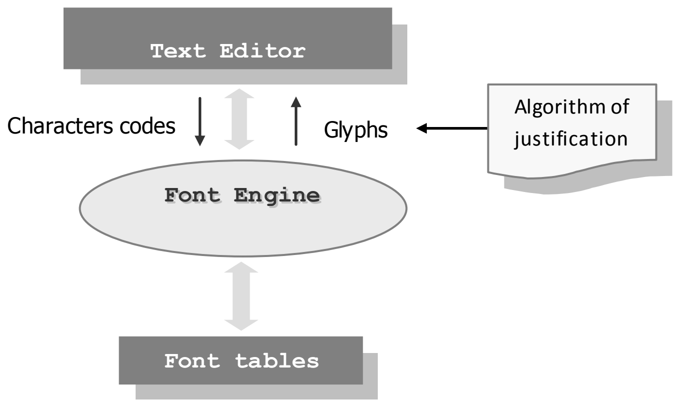
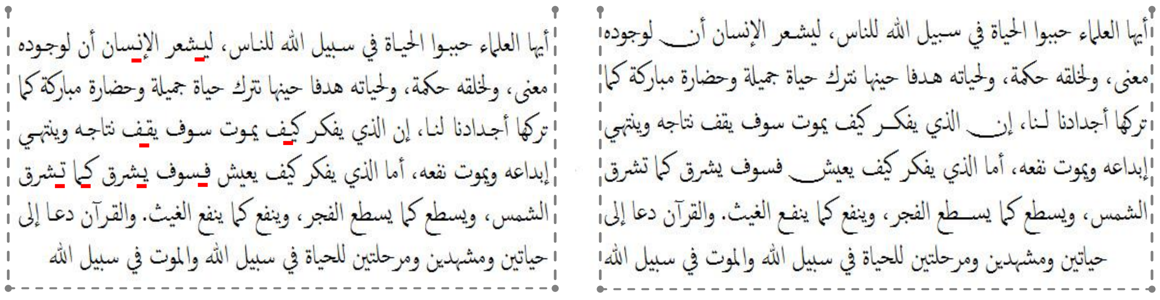


Fig. 7. Integrating our justification algorithm with the text editor 'neatpad'.

The justification of the text is based on measuring the width of the glyph before rendering them on the screen. Since the algorithm is implemented in the middle between the font engine and the application when the stream of glyphs is returned, this translates to a line wise justification of text. This means our algorithm justifies text at line level rather than at a paragraph level.

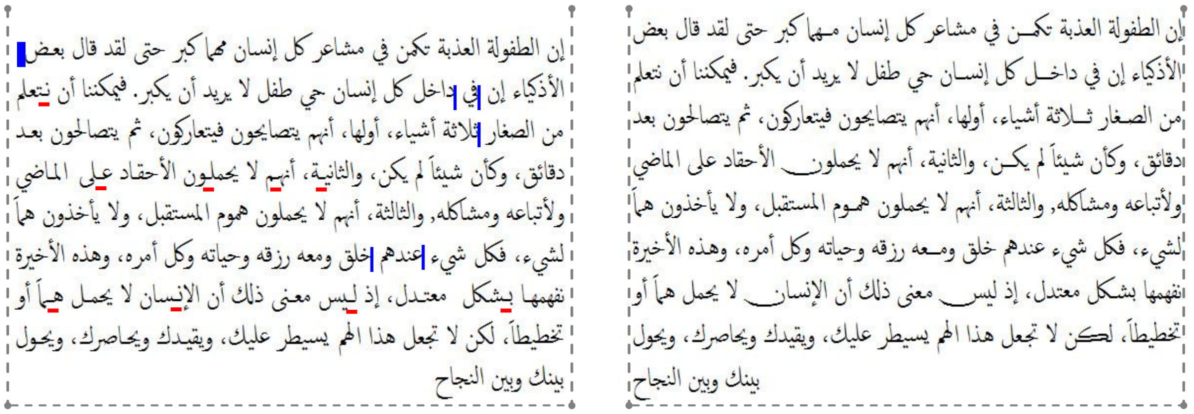
Next we go over some sample output of our algorithm. Each sample will be rendered by our justification algorithm and the standard built-in justification in MS Word. The first two samples (Fig. 8-9) are for plain text while the last one (Fig. 10) is for a vocalized text (full diacritical marking).


أيها العلماء حببوا الحياة في سبيل الله للناس، ليشعر الإنسان أن لوجوده
معنى، ولخلقه حكمة، ولحياته هدفا حينها تترك حياة جميلة وحضارة مباركة كما
تركها أجدادنا لنا، إن الذي يفكر كيف يموت سوف يقف نتاجه وينتهي
إبداعه ويموت نفعه، أما الذي يفكر كيف يعيش فسوف يشرق كما تشرق
الشمس، ويسطع كما يسطع الفجر، وينفع كما ينفع الغيث. والقرآن دعا إلى
حياتين ومشهدين ومرحلتين للحياة في سبيل الله والموت في سبيل الله

أيها العلماء حببوا الحياة في سبيل الله للناس، ليشعر الإنسان أنـــ لوجوده
معنى، ولخلقه حكمة، ولحياته هدفا حينها تترك حياة جميلة وحضارة مباركة كما
تركها أجدادنا لنا، إنـــ الذي يفكـر كيف يموت سوف يقف نتاجه وينتهي
إبداعه ويموت نفعه، أما الذي يفكر كيف يعيشـــ فسوف يشرق كما تشرق
الشمس، ويسطع كما يســـطع الفجر، وينفع كما ينفع الغيث. والقرآن دعا إلى
حياتين ومشهدين ومرحلتين للحياة في سبيل الله والموت في سبيل الله

Figure 8. A sample sermon text justified using standard MS Word (left) and in neatpad using our algorithm (right). All the red colored underlines mark improper rendering.

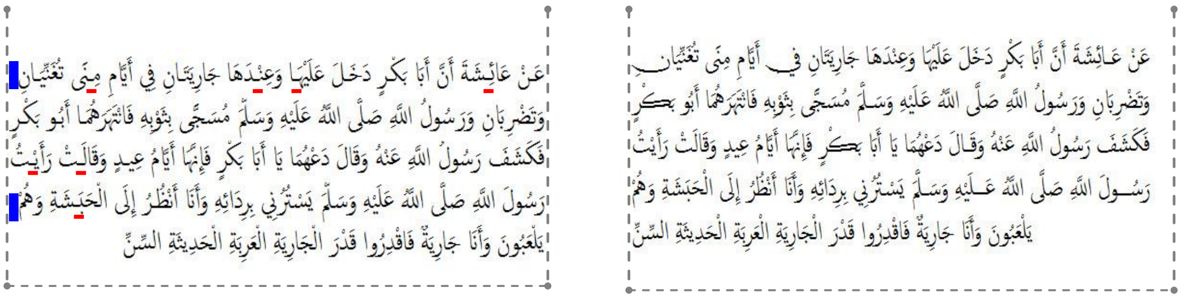
In Fig. 8 (left) we note wrongly place kashida by the standard MS Word justification algorithm. These are all underlined in red. For example in the top line we have a kashida between letters *yā'* followed by *shīn*, and a kashida between letter *nūn* followed by *sīn*. On the other hand our algorithm (Fig. 8 right) solved the problem of text justification in top line by using a wider glyph for the letter *nūn*.


إن الطفولة العذبة تكمن في مشاعر كل إنسان مهما كبر حتى لقد قال بعض
الأذكياء إن في داخل كل إنسان حي طفل لا يريد أن يكبر. فيمكننا أن نتعلم
من الصغار ثلاثة أشياء، أولها، أنهم يتصايحون فيتعاركون، ثم يتصالحون بعد
دقائق، وكأن شيئاً لم يكن، والثانية، أنهم لا يحملون الأحقاد على الماضي
ولأتباعه ومشاكله, والثالثة، أنهم لا يحملون هموم المستقبل، ولا يأخذون هماً
لشيء، فكل شيء عندهم خلق ومعه رزقه وحياته وكل أمره، وهذه الأخيرة
نفهمها بشكل معتدل، إذ ليس معنى ذلك أن الإنسان لا يحمل هماً أو
تخطيطاً، لكن لا تجعل هذا الهم يسيطر عليك، ويقيدك ويحاصرك، ويحول
بينك وبين النجاح

إن الطفولة العذبة تكمـــن في مشاعر كل إنسان مـــهما كبر حتى لقد قال بعض
الأذكياء إن في داخـــل كل إنســـان حي طفل لا يريد أن يكبر. فيمكننا أن نتعلم
من الصـــغار ثـــلاثة أشياء، أولها، أنهم يتصايحون فيتعاركون، ثم يتصالحون بعد
دقائق، وكأن شيئاً لم يكـــن، والثانية، أنهم لا يحملون الأحقاد على الماضي
ولأتباعه ومشاكله, والثالثة، أنهم لا يحملون همـــوم المستقبل، ولا يأخذون هماً
لشيء، فكل شيء عندهم خلق ومـــعه رزقه وحياته وكل أمره، وهذه الأخيرة
نفهمها بشـــكل معتدل، إذ ليس معنى ذلك أن الإنسان لا يحمل هماً أو
تخطيطاً، لكن لا تجعل هذا الهم يسيطر عليك، ويقيدك ويحاصرك، ويحول
بينك وبين النجاح

Figure 9. Another sample text justified using standard MS Word (left) and in neatpad using our algorithm (right). The red underlines and blue blocks mark places where rendering is incorrect.

In Fig. 9 (left) we see that MS Word failed to justify the top line. The blue colored block marks an extra space between the rightmost letter and the left margin. By inserting kashida at two places, our algorithm managed to eliminate that space in the top line.


عَنْ عَائِشَةَ أَنَّ أَبَا بَكْرٍ دَخَلَ عَلَيْهَا وَعِنْدَهَا جَارِيَتَانِ فِي أَيَّامِ مِنًى تُغَنِّيَانِ
وَتَضْرِبَانِ وَرَسُولُ اللَّهِ صَلَّى اللَّهُ عَلَيْهِ وَسَلَّمَ مُسَجًّى بِثَوْبِهِ فَانْتَهَرَهُمَا أَبُو بَكْرٍ
فَكَشَفَ رَسُولُ اللَّهِ عَنْهُ وَقَالَ دَعْهُمَا يَا أَبَا بَكْرٍ فَإِنَّهَا أَيَّامُ عِيدٍ وَقَالَتْ رَأَيْتُ
رَسُولَ اللَّهِ صَلَّى اللَّهُ عَلَيْهِ وَسَلَّمَ يَسْتُرُنِي بِرِدَائِهِ وَأَنَا أَنْظُرُ إِلَى الْحَبَشَةِ وَهُمْ
يَلْعَبُونَ وَأَنَا جَارِيَةٌ فَاقْدِرُوا قَدْرَ الْجَارِيَةِ الْعَرِبَةِ الْحَدِيثَةِ السِّنِّ

عَنْ عَـائِشَةَ أَنَّ أَبَا بَكْرٍ دَخَلَ عَلَيْهَا وَعِنْدَهَا جَارِيَتَانِ فِـي أَيَّامِ مِنًى تُغَنِّيَانِ
وَتَضْرِبَانِ وَرَسُولُ اللَّهِ صَلَّى اللَّهُ عَلَيْهِ وَسَلَّمَ مُسَجًّى بِثَوْبِهِ فَانْتَهَرَهُمَا أَبُو بَكْرٍ
فَكَشَفَ رَسُولُ اللَّهِ عَنْهُ وَقَـالَ دَعْهُمَا يَا أَبَا بَكْرٍ فَإِنَّهَا أَيَّامُ عِيدٍ وَقَالَتْ رَأَيْتُ
رَسُـــولَ اللَّهِ صَلَّى اللَّهُ عَـلَيْهِ وَسَلَّمَ يَسْتُرُنِي بِرِدَائِهِ وَأَنَا أَنْظُرُ إِلَى الْحَبَشَةِ وَهُمْ
يَلْعَبُونَ وَأَنَا جَارِيَةٌ فَاقْدِرُوا قَدْرَ الْجَارِيَةِ الْعَرِبَةِ الْحَدِيثَةِ السِّنِّ

Figure 10. A sample fully diacritized religious text justified using standard MS Word (left) and in neatpad using our algorithm (right). The red underlines and blue blocks mark improper rendering.

Fig. 10 is a fully diacritized text. We note that MS Word used kashida in a three letter word (منى) which violates the calligraphic rules. All the red underlines mark wrongly placed kashidas. In addition, we have two under filled lines (lines 1 and 4).

6. Conclusion

In this paper we presented an algorithm that justifies Arabic text using agreed on calligraphic rules. It is a two step process. In the first step we use glyph substitution through composing/decomposing of the ligatures. In the next step we resort to kashida to fill in the under filled lines. The use of kashida (stretching of letters) is again dictated by calligraphic rules which are preserved in what is known as kashida matrix. We implemented our algorithm and compared the its output of justified text with the standard justification as used by MS Word and found that our algorithm yield results closer to the calligraphers sense of properly justified text.

References

[1] محمد حسيني، عزالدين لزرق، ومحمد جمال الدين بن عطية. "علامات الشكل في المستند الإلكتروني العربي". *المؤتمر الدولي الرابع لممارسات علوم الحاسب باللغة العربية*. جامعة قطر، الدوحة، قطر. (1-4 ابريل 2008).

[2] عزالدين لزرق، "أبناط في معالجة النصوص العلمية العربية". *ندوة تقنية المعلومات والعلوم الشرعية والعربية*. جامعة الإمام محمد بن سعود الإسلامية، الرياض، السعودية. (6-7 مارس 2007).

[3] محمد اليعقوبي، عزالدين لزرق، ومصطفى بنوني."نحو تيبوغرافية ديناميكية لمحاذاة النص العربي". *المؤتمر الدولي الرابع لممارسات علوم الحاسب باللغة العربية*. جامعة قطر، الدوحة، قطر. (1-4 ابريل 2008).

[4] M.J.E. Benatia, M. Elyaakoubi, and A. Lazrek, “Arabic Text Justification”, *Proc. 2006 Annual Meeting, TUGboat*, 27(2): 137-146, 2006.

[5] D.M. Berry, “Stretching Letter and Slanted-baseline Formatting for Arabic, Hebrew and Persian with ditroff/ffortid and Dynamic PostScript Fonts”, *Software Practice and Experience*, 29(15): 1417-1457, 1999

[6] M. Elyaakoubi and A. Lazrek, “Arabic Scientific e-document Typography”, *Proc. 5th Int. Conf. on Human System Learning (ICHSL5)*, pp. 241-52, 2005.

[7] H. Fahmy, “AlQalam for Typesetting Traditional Arabic Texts”, *Proc. of 2006 Annual Meeting, TUGboat*, 27(2): 159-166, 2006.

[8] Y. Haralambous, “Infrastructure for High-Quality Arabic Typesetting”, *Proc. of 2006 Annual Meeting*, pp. 1001-1009, 2006.

[9] D.E. Knuth and P. MacKay, "Mixing right-to-left texts with left-to-right texts", *TUGboat*, 8(1): 14-25, 1987.

[10] A.M. Sherif and H.A.H. Fahmy, “Parameterized Arabic Font Development for AlQalam”, *Proc. EuroTeX 2007, TUGboat*, 29(1): 501-510, 2007.

[11] "Ibn Muqla". Available: http://calligraphyqalam.com/people/ibn-muqla.html.