Writing Arabic on a Nonlinear Curve

Firaas Antar

April 2024

Abstract

This project presents a novel approach for writing Arabic text along a nonlinear curve. Traditional methods rely on Latin languages as a basis; however, due to the nature of written Arabic, with its connecting letters, text rendering often struggles with non-linear paths, resulting in distorted connections. Our solution leverages SVG manipulation and Bézier curves to accurately position Arabic characters along the curve, maintaining smooth connections and aesthetic appeal.

1 Introduction

Arabic calligraphy is renowned for its elegance and beauty, characterized by fluid curves and intricate designs. However, rendering Arabic text along non-linear paths poses a significant challenge, as existing techniques often yield unsatisfactory results. This project aims to tackle this challenge by developing a method for accurately writing Arabic text along arbitrary curves. By achieving this goal, we not only enhance the aesthetic appeal of Arabic calligraphy but also expand the possibilities for creative expression in digital design and typography. In this paper, we present our approach to achieving this goal.

2 Methodology

Our methodology consists of several key steps aimed at achieving the goal of accurately writing Arabic text along a nonlinear curve.

2.1 Problem Definition

It is first necessary to precisely define the problem we are trying to resolve: given a word in Arabic and a defined curve, we will return an SVG file with the word written along the given curve.

2.2 Setup

In light of the visual importance and significant user interaction required for this project, we opted to utilize HTML and JavaScript as the primary technologies. This decision was driven by the dynamic nature of the project, requiring real-time rendering and user input processing. HTML provides the structural framework for the user interface, while JavaScript handles the interactivity and dynamic generation of SVG files.
Additionally, we will elaborate on the construction of the initial unmodified word. This step is crucial as it serves as the foundation for subsequent modifications and transformations in the code. Understanding how the original word is constructed will clarify many of the functions and techniques used to generate the new SVG file.

2.3 Understanding SVG Files

Before delving into our strategies for modifying SVG files, it's important to understand the structure and format of SVG files. SVG (Scalable Vector Graphics) is an XML-based vector image format commonly used for rendering graphics on the web. SVG files consist of elements such as paths, shapes, text, and gradients, defined using XML markup. For the purpose of the project, we will focus on paths, which define shapes through a series of commands and coordinates.

```
<svg width="200" height="200" viewBox="0 0 200 200" xmlns="
    http://www.w3.org/2000/svg">
  <path d="M50 50 C50 100 150 100 150 50 S250 100 250 50 l0
      100 Z" fill="none" stroke="black" stroke-width="2" />
</svg>
```

Listing 1: Example SVG Code

To properly follow the paper, it is imperative to understand how we interpret an SVG path. It is an assembly of points that are related to each other using different commands (e.g., C, S, l). It always takes the endpoint of the previous command as the starting point for the next one. The numbers following the command letters are the coordinates of the individual points needed to execute the command. Each command has two forms: uppercase and lowercase. These specify the reference of the coordinates. If it's uppercase, the coordinates are in absolute terms and are fixed to the overall grid. If the command is in lowercase, then it uses the endpoint of the previous command as the reference (coordinate 0,0), making it relative to the previous command. This distinction is a powerful feature of SVG and will be used throughout the paper.

2.4 Strategies

To achieve the desirable output, we have decided to pursue three distinct strategies, each offering unique insights into the problem.
The first strategy involves adjusting each point in the original SVG file to align

with the curve, resulting in a new SVG file with all points modified, while retaining the individual commands that comprise the SVG file.
The second idea is to segment the entire word into two types of segments: connectors and letters. Each type will be modified and adjusted differently.
The third approach is to create a dataset of training data consisting of words and curves, and to develop a machine learning algorithm capable of outputting a correct SVG file adapted to the curve.

3 Implementation

3.1 Creation of the word

3.1.1 From Mixed coordinates to Relative coordinates

Many SVG files representing letters contain a mix of absolute and relative commands. However, for ease of translating the letters, it is advantageous to convert all commands to relative format, with only the starting M command being absolute. This approach simplifies the process of moving the letter, as changing a single command adjusts the position of the entire letter.
To achieve this goal, we have developed a function, which transforms all commands from absolute to relative format. The function takes two parameters: the first is the string representation of the SVG path, and the second is an offset value. This offset allows for moving the letter horizontally, facilitating its manipulation in subsequent stages of the implementation. Furthermore, it returns the left and rightmost extremities of the given SVG path, which will be used later on to find the value of the offset.

3.1.2 Construction of the Word

After processing the user's input and retrieving the corresponding files for each letter, we compile a list of paths representing each letter. Beginning with the first letter, we apply the 'absolute to relative' function to obtain the string of relative commands and determine the left and rightmost points, which are then used to calculate the offset. The next letter is then passed through the same function just with the new updated offset. This will return a new path that has its first 'M' command adjusted to follow right after the first letter, and with its left and right most points we can calculate its width and add it to the offset. This process iterates, generating a list of paths that can be concatenated to form the complete word's path.

3.1.3 From Relative to Absolute

Since we aim to modify each point according to the curve, having the path in relative coordinates complicates this task. Therefore, we created a function capable of transforming a path from relative commands to absolute, intended for

use on the final path of the word.
This function takes a string representation of the path and returns a list of commands. Each command is a list where the first element is the command letter, followed by the points in absolute coordinates required to execute the command (e.g., M663 314 c0 -29 -22 -51 -51 -51 is transformed into [["M", 663, 314],["C", 663, 285, 641, 263, 612, 263]]). Since our focus is on individual commands and their points rather than the entire path, this format significantly enhances ease of use for subsequent computations.

3.2 Adjusting each point to the given curve

3.2.1 What type of curve are we using

While selecting the curve, we considered two primary requirements. Firstly, it needs to be easy to implement and adjustable for users dynamically. Secondly, we aimed for a curve that requires a modifiable number of points to describe it while offering versatility in its forms.

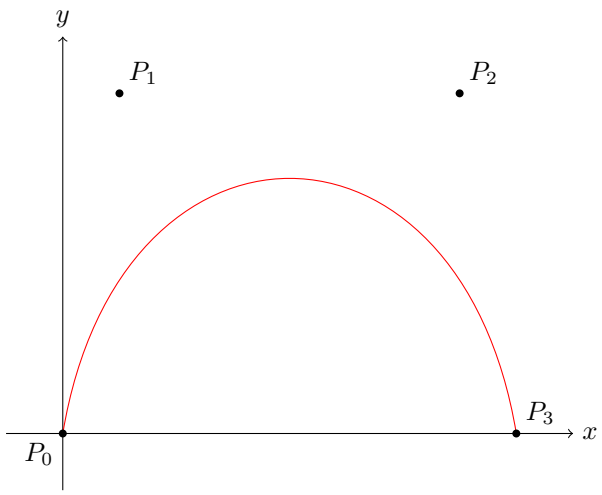
After careful consideration, we concluded that utilizing a Bézier curve would be the most appropriate choice. It fulfills all of our criteria and is already a supported command in SVG.

3.2.2 What are Bézier curves

Bézier curves are widely used in computer graphics and design to represent smooth curves. They are defined by a set of control points that influence the shape of the curve. For the purpose of this paper, we will focus on cubic Bézier curves, which are curves described by 4 control points and represented in SVG by the "C" command. A cubic Bézier curve is defined by four control points: P_0, P_1, P_2, and P_3. The curve starts at P_0 and ends at P_3, while P_1 and P_2 control the direction and shape of the curve. The parametric equations for a cubic Bézier curve are given by:

$$x(t) = (1-t)^3 PX_0 + 3(1-t)^2 t PX_1 + 3(1-t)t^2 PX_2 + t^3 PX_3$$
$$y(t) = (1-t)^3 PY_0 + 3(1-t)^2 t PY_1 + 3(1-t)t^2 PY_2 + t^3 PY_3$$

where $t \in [0,1]$ and PX_0, PY_0 are the x and y-coordinates of P_0 respectively. Let's consider an example where $P_0 = (0,0)$, $P_1 = (0.5,3)$, $P_2 = (3.5,3)$, and $P_3 = (4,0)$. We can plot the cubic Bézier curve using these control points.



A useful property of Bézier curves is that by constructing an n-level Bézier curve, you are using two $n-1$ Bézier curves, each made up of smaller ones. The important information is that the line passing through the two points at parameter t of the $n-1$ Bézier curves is the tangent of the point at parameter t of the n Bézier curves. This property will be useful later on in certain calculations.

3.2.3 The Format of the SVG Path of the Curve

The base of the path consists of an M command followed by a C command. For example, using the graph above: $M0,0\ C0.5,3,3.5,3,4,0$. Here, P_0 is given from the M command, and P_1, P_2, and P_3 are given from the C command.

To add more points to the curve, making it more flexible, we have decided to append S commands at the end of the path of the curve. These S commands are called smooth curves and they require two points as parameters. They are still cubic Bézier curves that use the previous command to construct P_0 and P_1, and use the points given in the S command for P_2 and P_3. P_0 is simply the last point in the previous command; however, P_1 is calculated using the two last points of the previous command, subtracting the second-to-last from the last and adding it to the last point. Thus, we end up with four points (following the example):

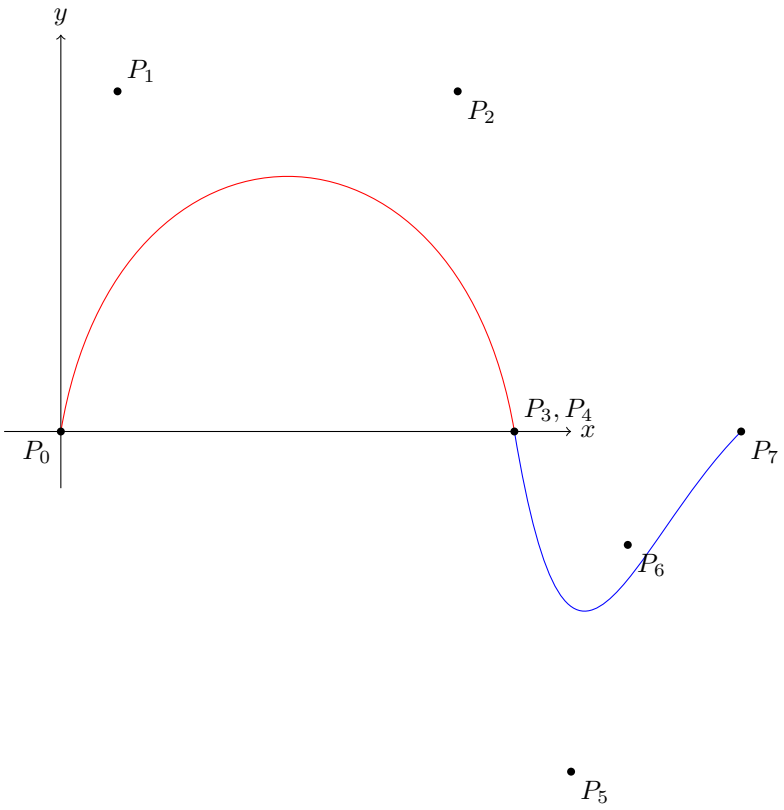
If we have $M0,0\ C0.5,3,3.5,3,4,0\ S5,-1,6,0$, we will then have two Bézier curves. The first one having:

$$P_0 = (0,0),\ P_1 = (0.5,3),\ P_2 = (3.5,3),\ \text{and}\ P_3 = (4,0)$$

And the second one having:

$P_4 = (4,0)$, $P_6 = (5,-1)$, $P_7 = (6,0)$, and $P_5 = ((4-3.5)+4, (0-3)+0) = (4.5,-3)$

The following graph illustrates the resulting Bézier curves formed by the given SVG path:



In this example, we have appended just one S command, but we can add as many as we like, making the curve smooth and flexible, and composed of many segments that are cubic Bézier curves.

3.2.4 Transformation function

The strategy involves conceptualizing a line passing horizontally through the center of the connection. Then, for every point on the SVG path of the word, we calculate both the perpendicular distance from that line and the position of the point along the line. This method establishes a new coordinate system that can be applied to any line, regardless of its linearity. By determining how far along the curve we need to proceed, we can ascertain the normal line at that point and, consequently, derive the new coordinates for the point.

3.2.5 Adjusting the individual points

After passing the string of the path of the entire word through the function 'relative_to_absolute', we now have a list of commands with their respective points. However, this format is not very useful to us since we are interested in

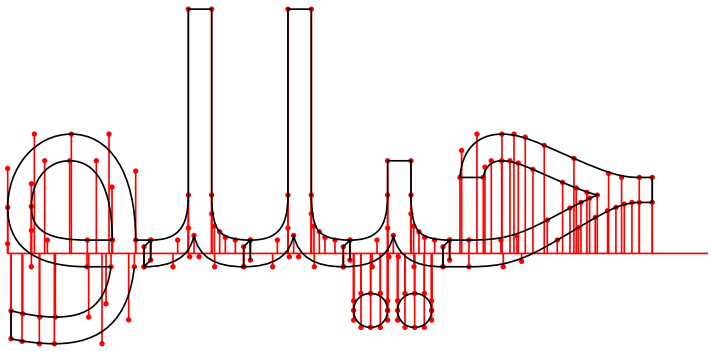


Figure 1: Points of SVG Word Path

the individual points. Thus, we decided create a table and add each point with its characteristics.

The table we have created has 5 columns:

- (x, y): This column represents the original coordinates of the point.
- d: As discussed earlier, there is an imaginary line passing through the Arabic word. Since this line is straight, the function describing it is a constant k. The distance d of each point from this line is simply calculated as: $d = |k - y|$.

```
    function get_distances(lst, y = 941){
            let distances = []
            for(var i = 0; i < lst.length; ++i){
                for (var j = 2; j < lst[i].length; j += 2){
                    distances.push(parseFloat(lst[i][j]) - y)
                }
            }
            return distances
        }
```

Listing 2: Get the distance of each point to the line

- commandNumber: In parallel, a second list will be created containing all the commands in order. This column represents the index of the command to which they belong.
- t: The t value represents the parameter passed to the Bézier curve function and ranges between 0 and 1. It is calculated by dividing the x value of the point by the total length of the space over which the word is written.

In the construction process, a variable called 'percentage' is introduced to slide the word along the curve. The 'getT' function computes the t values:

```
    function getT(lst){
            var percentage = document.getElementById("shift_t").value
            var lr = getLeftandRightMost(lst)
            var d = lr[1] - lr[0]
            var ratio = d/4000

            var toAdd = (3900 - lr[1])*percentage/4000
            var tValues = []
            for(var i = 0; i < lst.length; ++i){
                for(var j = 1; j <lst[i].length; j += 2){
                    var temp = (((lst[i][j] - lr[0]) * ratio)/d)

                    tValues.push(temp + toAdd)
                }
            }
            return tValues
        }
```

Listing 3: Get the distance of each point to the line

- SegmentNumber: However, since we have multiple Bézier curves we need to calculate the t value of each point on its respective segment.Thus, we need to keep track what Bézier curves the point as associated to.

This is done using a function we called 'formatAbsoluteList', that takes in the list of commands for the word and the list of commands of curves we are working with. It then returns a table as described above and a list of all commands.

Once we have finished the table, we can finally start computing. The first thing we need are the points that make up the Bézier curve segment on which it is located and the true value of t on the segment. Using the segment number and the total number of segments, we can determine the range in which the point is located.

For t in segment i: $t \in \left[\frac{i-1}{n}, \frac{i}{n}\right]$. We can then transform this to have the range between 0 and 1:

$$\begin{aligned} \frac{i-1}{n} \leq t < \frac{i}{n} \\ 0 \leq t - \frac{i-1}{n} < \frac{i}{n} - \frac{i-1}{n} \\ 0 \leq t - \frac{i-1}{n} < \frac{1}{n} \\ 0 \leq nt - i + 1 < 1 \end{aligned}$$

This formula now allows us to use the points describing the curve to calculate three points: P_3, the point on the curve, and $P_{0,2}$ and $P_{1,2}$, the points on the tangent of the curve at P_3.

$$
\begin{aligned}
P_3(t) &= (A, B) \\
A(t) &= (1-t)^3 PX_0 + 3(1-t)^2 tPX_1 + 3(1-t)t^2 PX_2 + t^3 PX_3 \\
B(t) &= (1-t)^3 PY_0 + 3(1-t)^2 tPY_1 + 3(1-t)t^2 PY_2 + t^3 PY_3 \\
P_{0,2}(t) &= (1-t)^2 P_0 + 2(1-t)tP_1 + 2t^2 P_2 \\
P_{1,2}(t) &= (1-t)^2 P_1 + 2(1-t)tP_2 + 2t^2 P_3
\end{aligned}
$$

We calculate the slope of the normal line using $P_{0,2}$ and $P_{1,2}$:

$$N = \frac{x_{p0,2} - x_{p1,2}}{y_{p1,2} - y_{p0,2}}$$

With these points and the distance d, we are able to derive a formula for calculating the x-coordinate of the adjusted point.

We first find the equation of the normal line:

$$
\begin{aligned}
B &= N \cdot A + b \\
b &= B - N \cdot A \\
y &= N \cdot x + B - N \cdot A
\end{aligned}
$$

Then, we can use geometry to find the x and y coordinates:

$$
\begin{aligned}
d^2 &= (A - x)^2 + (B - y)^2 \\
d^2 &= (A - x)^2 + (B - (N \cdot x + B - N \cdot A))^2 \\
d^2 &= (A - x)^2 + (N \cdot A - N \cdot x)^2 \\
d^2 &= A^2 + x^2 - 2 \cdot A \cdot x + (N \cdot A)^2 + (N \cdot x)^2 - 2 \cdot A \cdot N^2 \cdot x \\
0 &= A^2 + x^2 - 2 \cdot A \cdot x + (N \cdot A)^2 + (N \cdot x)^2 - 2 \cdot A \cdot N^2 \cdot x - d^2 \\
0 &= \left(N^2 + 1\right) \cdot x^2 - 2 \cdot A \cdot \left(N^2 + 1\right) \cdot x + A^2 \cdot \left(N^2 + 1\right) - d^2
\end{aligned}
$$

Where:

$$
\begin{aligned}
a &= \left(N^2 + 1\right) \\
b &= -2 \cdot A \cdot \left(N^2 + 1\right) \\
c &= A^2 \cdot \left(N^2 + 1\right) - d^2 \\
x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{aligned}
$$

This now gives us the x-coordinate of the adjusted point. However, we need to determine whether to subtract or add the square root. There are three factors

that can influence our decision: if we are writing upside down, the sign of d, and the sign of N.

$$\text{if } d < 0 \text{ and } N < 0, \text{ then } +$$
$$\text{if } d < 0 \text{ and } N > 0, \text{ then } -$$
$$\text{if } d > 0 \text{ and } N < 0, \text{ then } -$$
$$\text{if } d > 0 \text{ and } N > 0, \text{ then } +$$

If $x_{p0,2} > x_{p1,2}$ that means that we are writing upside down and thus all these scenarios are flipped (e.g., $d < 0$ and $N < 0$ then $+$).

y is now easily calculated using the equation of the normal line, and we have successfully adjusted a point onto the given curve. This process is done for each point and then, together with commanNumber and the list of commands, we are able to reconstruct a new path of the word adjusted to the line.

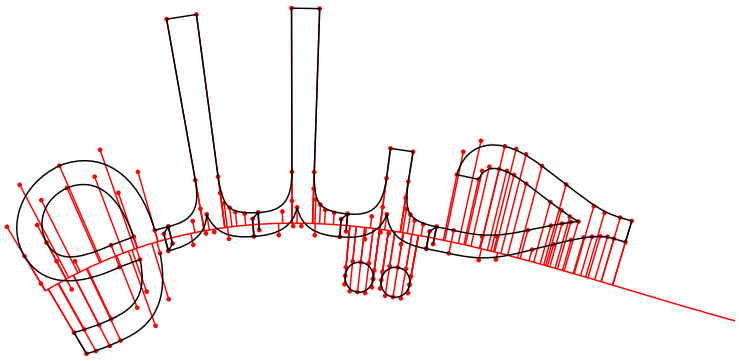


Figure 2: Adjusted Points of SVG Word Path

3.2.6 Restrictions

Although this strategy helped us address the connection issue, it introduced significant distortion due to the parameterization of Bézier curves. This resulted in undesirable effects, such as stretching away from the middle line and inconsistent outputs for the same word on seemingly identical curves. These problems prompted us to explore a second strategy that effectively resolved these issues.

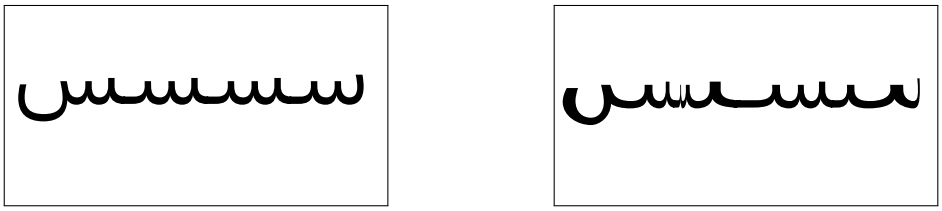


Figure 3: Unwanted Distortion

3.3 Segmentation of the Letters and adaptable fitting to the curve

3.3.1 How are we segmenting the letters

After evaluating the results of the previous strategy, we observed that only certain parts of the letters need modification, while the rest should be simply transposed. We decided to divide each letter into two segments: the core, which we want to keep intact and only rotate or slide, and the extremities or connectors. The extremities are the parts that connect one letter to another or, if there's no connection, the space between letters. We aim to fit this part perfectly to the curve and adjust its distortion to achieve a visually appealing result.

3.3.2 Finding the cores and segments

Due to the nature of the letters, there is an easily identifiable pattern that signifies the edge. This was used to segment the letter, and we accomplished this by formatting the SVG code in two different ways. The core is written with spaces (e.g., M250 500 l100 2000), while the connector is written with commas (e.g., M250,500,L100,2000). This allows us to easily identify the different segments in the code and manipulate them accordingly.

3.3.3 Rotating Cores and Connectors Based on Curve Intersections

Since we need to adjust to the curves, we must know where and by how much we need to rotate the cores and connectors. To achieve this, we can use the length of the core and the length of the connector to draw circles and find their intersections with the curve. The idea is that we take the length of the first core and draw a circle with its center at the beginning of the curve and a radius equal to the length of the core. At the intersection with the curve, we will get a point that, along with the center of the circle, will give us a line on which the core needs to be rotated. This new point will now become the center of the next circle, which will have a radius equal to the length of the connector. Its intersection point will become the center of the next circle, which will take the length of the second letter. This process is then repeated for all the letters.

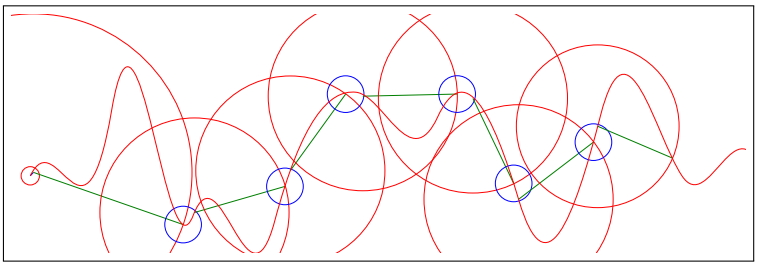


Figure 4: Circles and their Intersections Drawn According to the Curve.
Legend: Red: core circle, Blue: connection circle, Green: core line.

3.3.4 Aligning Letters Using Cubic Bézier Curves

After acquiring the lines associated with each letter, we can now adjust the letters to their corresponding lines, effectively rotating and translating them to the correct locations. Building on our previous strategy, we can use its function to achieve this goal. However, for the function to work as intended, it requires a cubic Bézier curve in SVG format rather than a simple line. This is easily done by creating a Bézier curve, with the first point of the line as the starting point and the second as the ending point. The intermediate control points are calculated to ensure that all points are equidistant from their neighbor. With this newly constructed curve and the function from our previous strategy, we can apply this process to each letter, aligning them at the correct angle and position.

3.3.5 Edge Detection and Pattern Recognition in Arabic Fonts

Now that we have placed the letter at the correct angle and location, the next step is to identify where the edges of the letters are, in order to determine where to place the connection. For most Arabic fonts, there is a specific pattern at the edges that symbolizes whether it is a left or right edge. Each font may differ in the patterns they use; however, in our example, we used a triangle cut-out, as seen in the attached image. Due to the various ways this pattern can be drawn with SVG, we had to account for these differences in order to identify all edges correctly, regardless of the variation.
The function we created is able to identify the edges of each letter and how many there are. It then returns the coordinates of the red points (as seen in the image below). This function was designed with the understanding that the pattern we found is not universal, and it can be modified to detect any pattern one wishes to use for identifying edges.

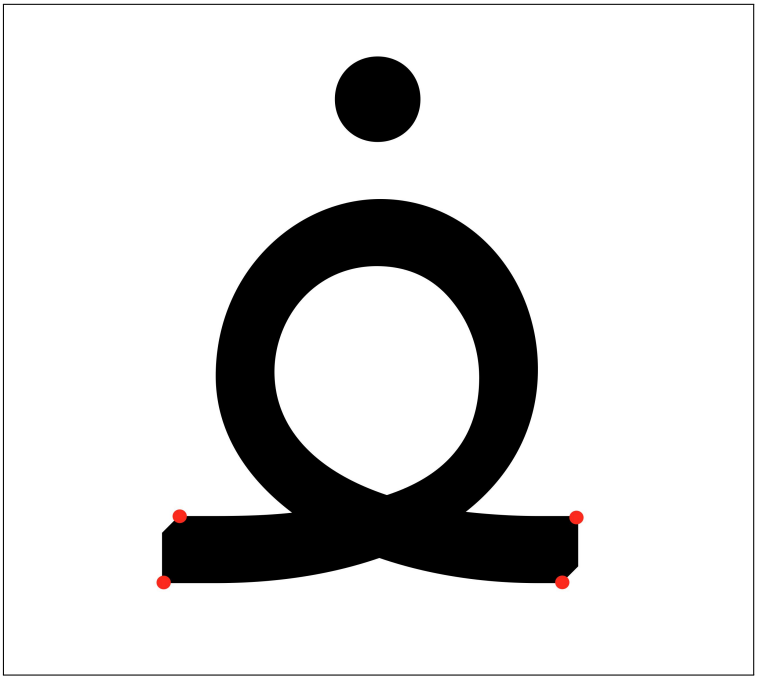


Figure 5: A letter showing the left and right edge patterns.
Legend: Red: Returned coordinates

3.3.6 creating the connection

From the points we acquired in the previous step, we decided to create the connection by constructing a structure that relates those points. The two points located on the same edge are connected by a simple line, while the two lower points on different letters are joined by a cubic Bézier curve, as are the upper points. To achieve this, we needed to identify two additional points for each curve, which serve as control points (weights).

We will use the lower curve as an example to explain our methodology. The lower point on the left letter and the lower point on the right letter lie on lines parallel to the line described in section 3.3.3. Using this information, we extended both lines from the left and right sides toward the center of the connection. We then identified one point on the left line and one point on the right line such that the intervals between all four points were equal.

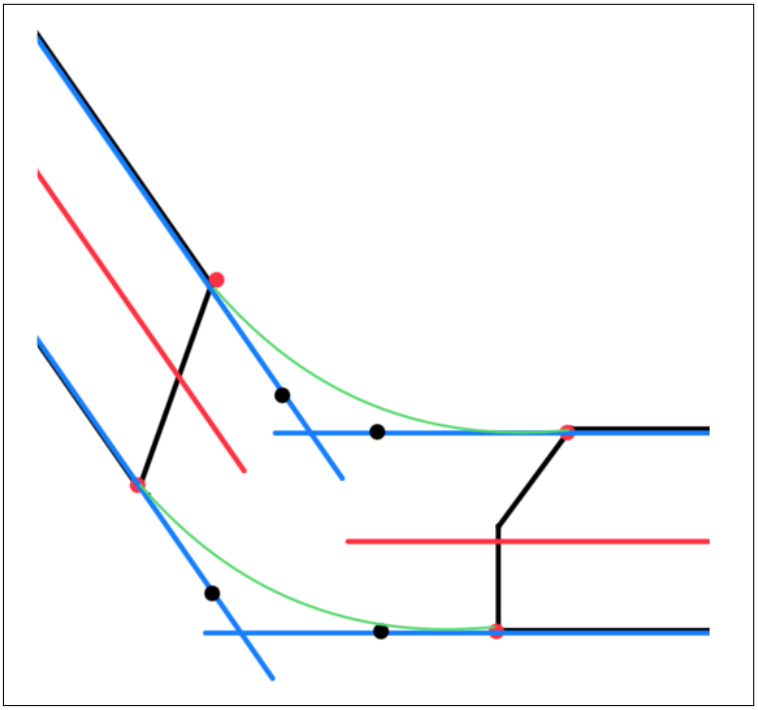


Figure 6: Construction of the connectore element.
Legend: Lines: Lines from section 3.3.3; Points: Points from section 3.3.5; Lines: Lines parallel to the red ones passing through the red points; **Points**: Points on the blue lines, so that the distance between all 4 upper, and lower, points is equal; Curve: Curve created by using the red points as start and end points, and using the black points as control points.

3.4 Results

With this final version, we are now able to write Arabic on a nonlinear curve without sacrificing the structure of the individual letters while maintaining a seamless connection. It is theoretically capable of adapting to any font that uses the same pattern for the edges. For fonts with different patterns, they can be adjusted with some simple modifications to one function.

3.5 Conclusion

In conclusion, the advancements made thus far represent a significant step forward in expanding the scope of Arabic fonts within the digital realm. However, it is important to recognize that this is just the beginning of our journey. While our methods have demonstrated the potential for writing Arabic on nonlinear curves, they are limited by the time required to calculate intersections. As more letters are added to a word, the rendering wait times can increase significantly,

potentially hindering the user experience. This is due to the calculation process involved in finding intersections between the circles and the curve. Initially, we developed an equation that combines the equation of a circle with parametrized functions, resulting in equations of order 6. However, we later discovered that circles can be replaced with cubic Bezier curves. This insight allows us to replace the previous methodology with one that seeks the intersection of two cubic Bezier curves, reducing the equation's complexity by three orders. This should be the strategy moving forward.

Additionally, the current implementation does not support writing upside down due to complications in identifying the correct roots, as discussed in section 3.3.3. This limitation further suggests that, while our approach shows promise, it is not yet commercially viable.

Despite these challenges, we believe these issues can be addressed through ongoing research and development. Future work could involve optimizing the algorithms for intersection calculations to reduce wait times, as well as refining our methods to accommodate different text orientations. By addressing these limitations, we can enhance the functionality and applicability of our approach, paving the way for more versatile and user-friendly Arabic font rendering in digital applications. Ultimately, our findings provide a foundation for further exploration and innovation in the field of digital typography.