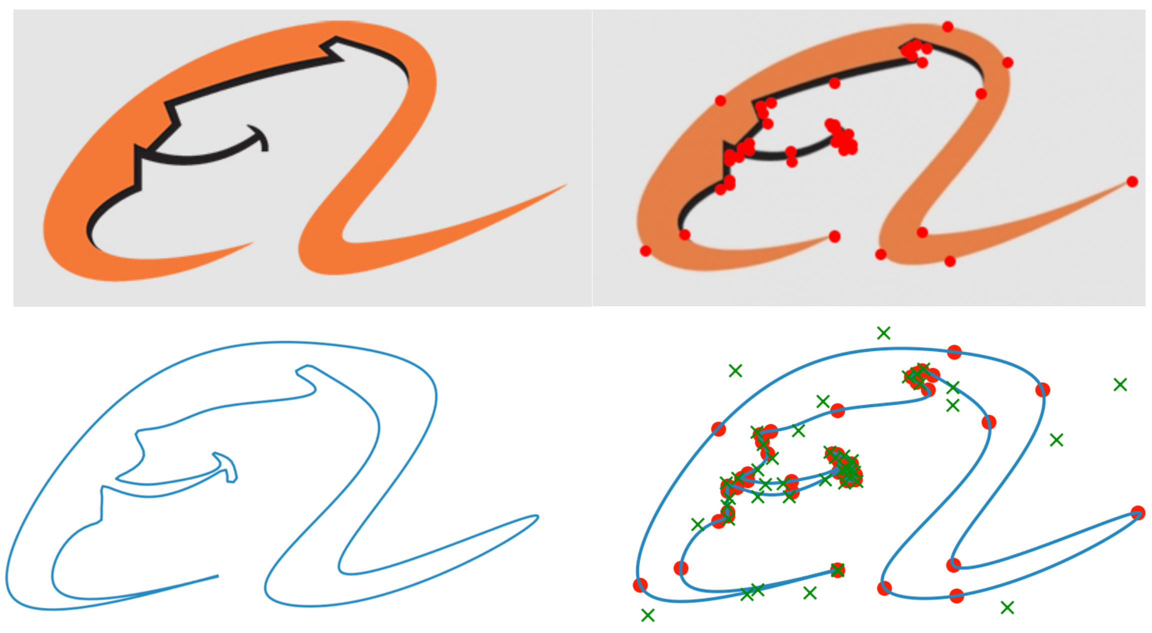
CE7453 Numerical Algorithms

Assignment 1: Cubic B-Spline Interpolation

Pung Tuck Weng (G1803000G) – IGS-Alibaba Talent Programme



Top-Left to Bottom-Left (clockwise): Alibaba Logo, Logo with Input Coordinates, Interpolated Cubic B-Spline & Control Points, Final Interpolated Curve

Introduction

The B-spline interpolation program was written in Python 3, due to its usefulness as a programming language and ease of use. The program was written in PyCharm IDE. The script was written functionally, breaking each part of the output into separate functions.

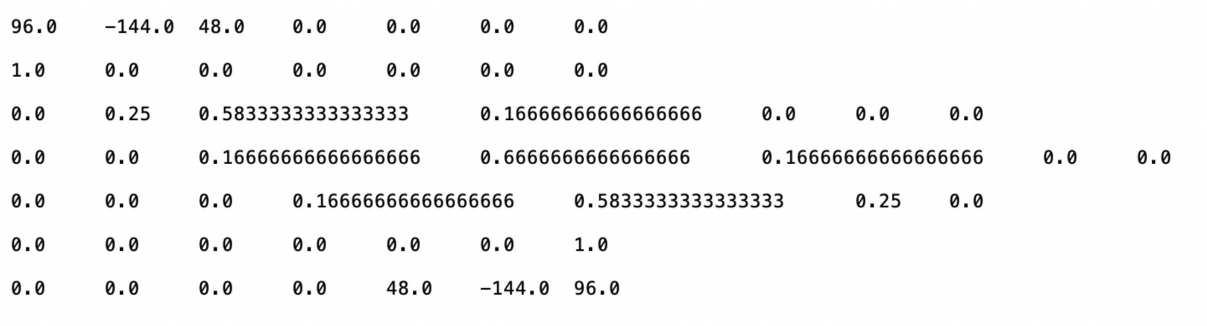
Following the cubic b-spline interpolation steps laid out by the de Boor algorithm, the program first opens the input text file and parses the relevant interpolated coordinates into a list. Then, chord-length parameterization was done to generate the knots vector. The knots vector was then padded with zeros and ones corresponding to the degree of the interpolated b-spline, which is three in the case of cubic b-splines. Following the required output format, the knot vector is then appended into the variable 'spline' which will be eventually saved as an output 'cubic.txt'.

The example image you see at the top of this page is an interpolation of the Alibaba logo, as a fun test example of the usefulness of B-splines in vectorizing (compressing) the image rather successfully.

Linear System

The program has a function get_N_value which takes in the knot vector and information surrounding it, and outputs the N value using the formula given. The function checks which knot vector interval the input u is within, then selects the appropriate function; where the denominator for the specified function is zero, the next interval is selected (particularly for cases where the knot vector falls within both intervals). The intent of this function was to recreate the $N_i^3(u)$ formula which is extensively used generating the N matrix in the linear system, as well as in converting the control points into coordinates for plotting the b-spline.

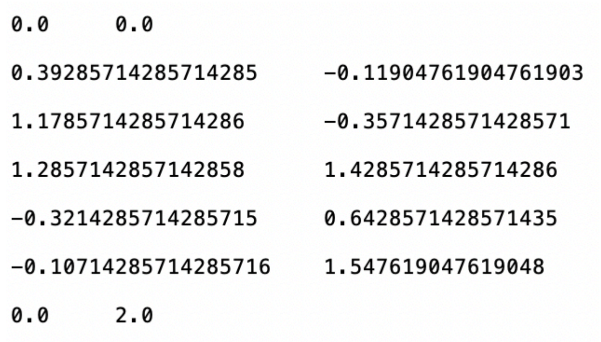
By observation, the N matrix only has 3 values per row; hence the script used to generate the N matrix first instantiates a zero matrix of N+3 by N+3 (where N is the number of coordinates minus 1), then the get_N_value function is called for the appropriate row and column in the N matrix to calculate the corresponding values. The last non-trivial work in populating the N matrix was in the last row (the endpoint conditions) where the differentiated value is required; here, a symbolic calculus library (SymPy) was invoked to aid in the calculation.



```
96.0    -144.0  48.0    0.0     0.0     0.0     0.0
1.0     0.0     0.0     0.0     0.0     0.0     0.0
0.0     0.25    0.5833333333333333      0.16666666666666666     0.0     0.0     0.0
0.0     0.0     0.16666666666666666     0.6666666666666666      0.16666666666666666     0.0     0.0
0.0     0.0     0.0     0.16666666666666666     0.5833333333333333      0.25    0.0
0.0     0.0     0.0     0.0     0.0     0.0     1.0
0.0     0.0     0.0     0.0     48.0    -144.0  96.0
```

N-matrix for test-case example (input.txt)

Once the N matrix is successfully filled, we can then solve the linear equation using the numpy.linalg.solve() function, generating the necessary D matrix.



```
0.0     0.0
0.39285714285714285     -0.11904761904761903
1.1785714285714286      -0.3571428571428571
1.2857142857142858      1.4285714285714286
-0.3214285714285715     0.6428571428571435
-0.10714285714285716    1.547619047619048
0.0     2.0
```

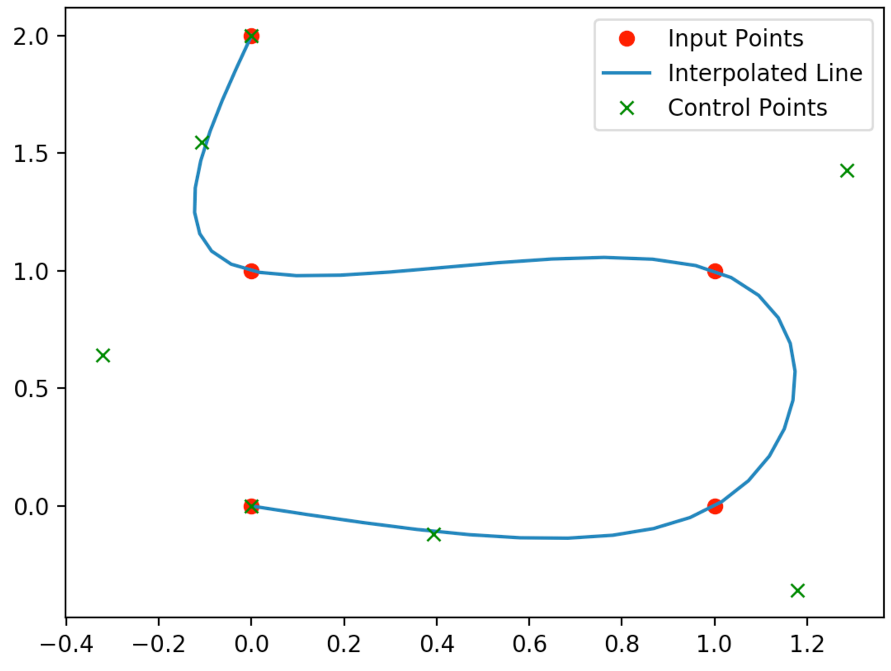
Corresponding D-matrix for test-case example (input.txt)

Once the D Matrix is generated, I followed the definition of b-splines to sum across the points multiplied by the knot vectors' influence (calculated from the N matrix), in the formula below:

$$r(u) = \sum_{i=0}^{n} P_i N_i^k(u), \qquad u \in [u_k, u_{n+1}]$$

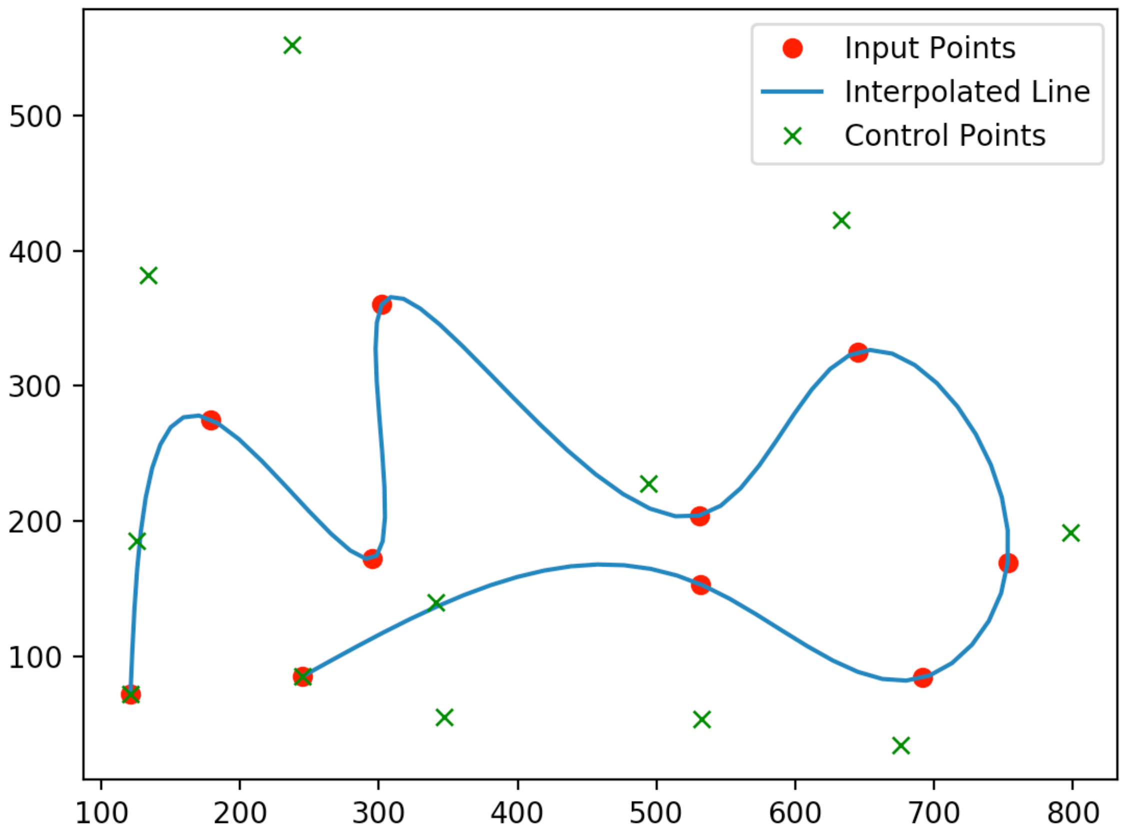
To make the interpolated curve as smooth as possible, two strategies was used. Firstly, chord length parameterization was used over linear parameterization; secondly, the number of points in the interpolated curve scales proportionally to the length of the generated d matrix by a factor of 10. Thanks to the speed and efficiency of the program, even for a curve with 45 control points (450 interpolated coordinates), the curve was generated instantaneously.

Examples



Default Input

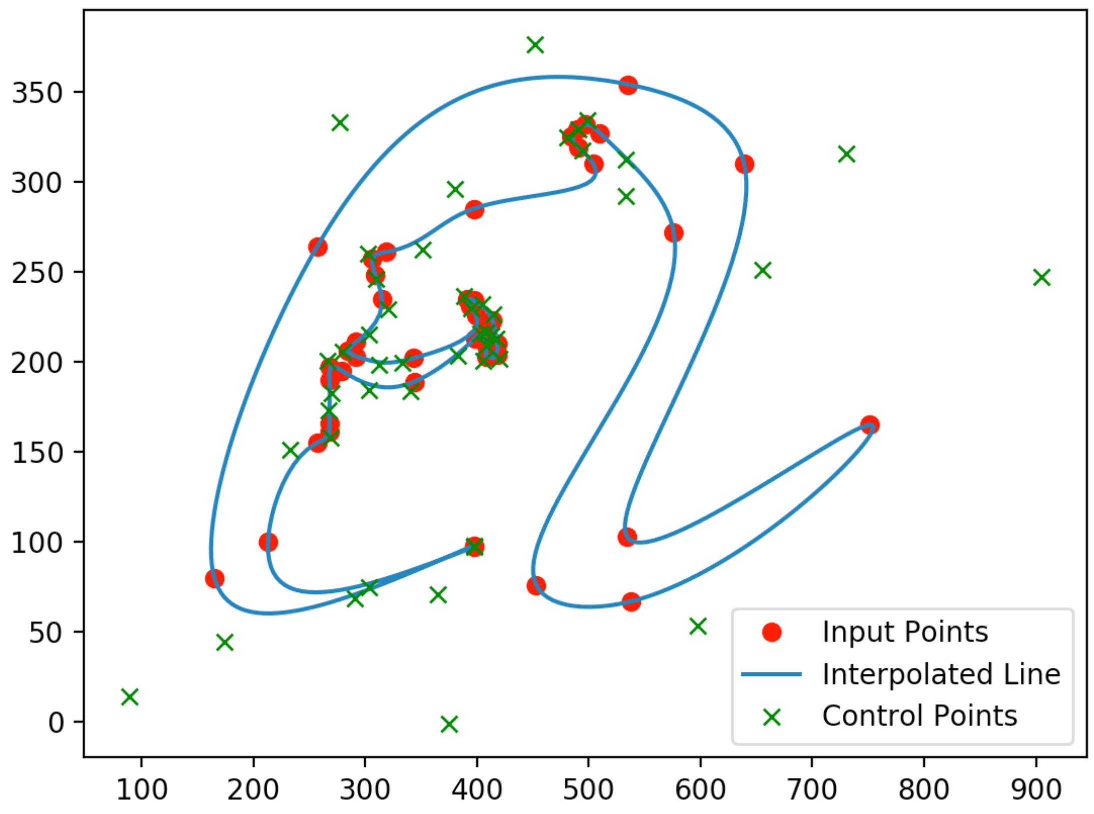
In this default input of 5 points, the program interpolates well, where the start and end control points coincide with the first and last coordinated as expected. This is a trivial example, and more rigorous tests can be applied to examine if the program works well.



Test Input #1

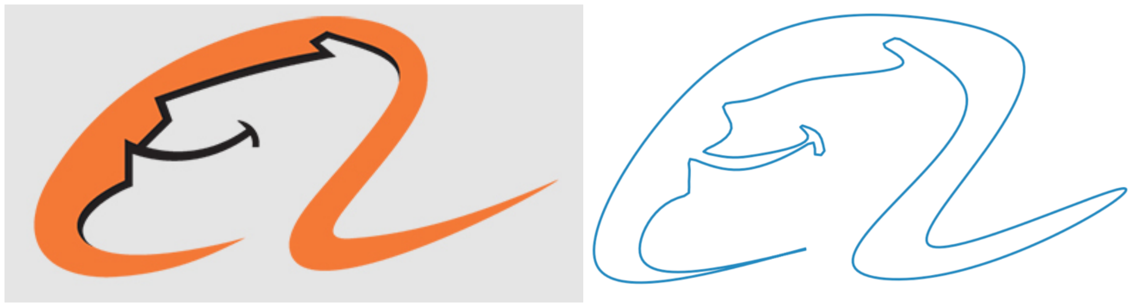
In this slightly more rigorous example, ten random input points was generated to see if the program can handle larger values of both the magnitude of individual input coordinates, as well as an increased number of coordinates (and a corresponding larger linear system to solve). In this case, the program successfully interpolates the points well, with a smooth B-spline (the blue line) generated that passes through all input points (represented with red dots).

From the above example, it can be shown that the program is working well. The next example will then be used to illustrate the usefulness of B-splines, particularly as a compressive algorithm in generating vector images (and in its more ubiquitous form, in generating font types).



Test Input #2

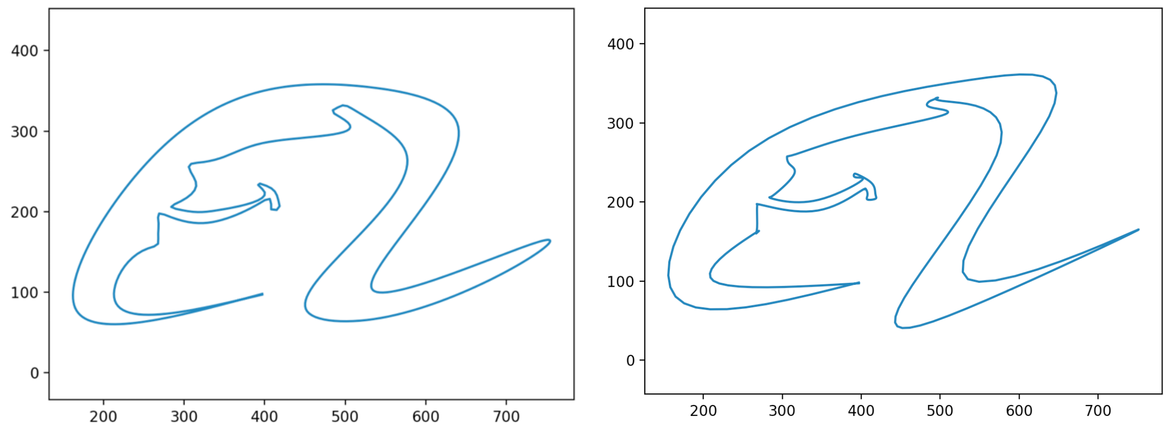
In this test input of 46 points, the B-spline interpolation program was used to recreate the logo of a Chinese Internet company. First, the set of coordinates to-be-interpolated was generated from overlaying above an image of the logo. After running the B-spline algorithm, the outline of the logo was generated, and a side-by-side comparison of the logo can be shown below.



Left: Alibaba Logo, Right: Interpolated B-Spline

Looking at the fidelity of the generated curve, it can be shown that cubic B-splines can very successfully be used to generate vector images commonly found in digital logos. This can form a very compact compressive algorithm over saving the logo as a raw image, as only 46 coordinates was used (compared to likely over a million RGB pixel values in raw images).

Parameterization Analysis



Left: Chord-length parameterization, Right: Uniform parameterization

Out of curiosity, I wanted to empirically test if the chord-length parametrization was indeed superior to uniform parameterization. After all, computing the chord-length parameterized form of knot vectors requires traversing the input vector twice to both compute the total distance of the linearly interpolated curve, as well as to compute the successive Euclidean distances between each point; this is much more intensive then simply generating a linear uniformly-spaced vector of equal length to omit traversing the input vector.

The images above show the outcome of both parameterization methods: with chord-length parameterization, we see that the curves are indeed smoother where input coordinates are significantly closer together. This can be seen in the smooth outputs of the intricate parts of the logo, such as the edge of the mouth and hair sections of the face. The angle of curvature of the generated curve is also more accurate in chord length parameterization (compare using original logo in previous page), whereas the uniform parameterization results in too jagged and sharp turning points in the logo. Overall, it can be shown that chord-length parameterization is indeed superior, and the slight increase in computation is generally justified.

Conclusion

In conclusion, the B-spline algorithm has been implemented successfully, based on the expected inputs and outputs of the API. The report also shows the usefulness and efficiency of B-splines in recreating images, and it had been thoroughly enjoyable to have learnt much more about them.