**Bezier Interpolation**

Copied from <https://towardsdatascience.com/b%C3%A9zier-interpolation-8033e9a262c2> to retain the original content without link dependency.

**Cubic Bezier Curves**

The goal is to fit ***n+1*** given points ***(P0, …, Pn)***. To fit these points, we are going to use one cubic Bezier curve (4 control points) between each consecutive pair of points.

Diagram

Description automatically generated with low confidence

fig. 1

In this figure, ***G0***, ***G1,*** and ***G2***are three different cubic Bezier curves that start and end at ***(P0, P1)***, ***(P1, P2)***, and ***(P2, P3)*** respectively. Since any Bezier curve always starts and ends at the first and last control points, we are left with 2 control points for each curve that we will have to find so that the resulting line looks smooth.

The general equation of the cubic Bezier curve is the following:

Where ***K***are the 4 control points. In our case, ***K0*** and ***K3***will be two consecutive points that we want to fit (e.g. ***P0-P1***, or ***P1-P2***, etc.), and ***K1*** and ***K2***are the remaining 2 control points we have to find.

**Problem Setup**

Given that we have ***n+1***points to fit, we will use a cubic Bezier curve to fit each consecutive pair of points. We denote **Γi**the Bezier curve that fits ***Pi*** to ***Pi+1***:



eq. 2

Where ***ai*** and ***bi***are left to find. Notice that there are ***n*** curves.

If we want the final curve to be smooth, we need to ensure that the transition between **Γi**and**Γi+1**is smootharound ***Pi+1***. In other words, that the curvature of **Γi**matches the curvature of**Γi+1**around***Pi+1.***Mathematically, this means respecting the following conditions:

Shape

Description automatically generated with medium confidence

eq. 3, 4

We need to find all the ***ai*** and ***bi***. Since we have one pair of them in each Bezier curve, and since we have ***n***curves, we need to find ***2n***variables. However, here we have ***2(n-1)*** equations. We are missing 2 equations to solve the system. Therefore, we impose the following (arbitrary) boundary conditions:

Shape

Description automatically generated with medium confidence

eq. 5, 6

**Write the system**

Before solving the system, we need to calculate the first and second derivatives of **Γi**and write the system down.



eq. 7

and,



eq. 8

**Inject the equations**

Injecting eq. 7 into eq. 3:

Shape

Description automatically generated with medium confidence

eq. 9

Injecting eq. 8 into eq. 4:

Shape

Description automatically generated with medium confidence

eq. 10

Injecting eq. 8 into eq. 5:

Shape

Description automatically generated with medium confidence

eq. 11

Finally, injecting eq.8 into eq. 6:

Shape

Description automatically generated with medium confidence

eq. 12

**Solve the system**

To sum up, we have the following ***2n*** equations:

Shape

Description automatically generated with medium confidence

eq. 9, 10, 11, 12

To solve the system are going to eliminate all the ***bi***by injecting eq. 9 into eq. 10, 11, 12. Using eq. 9:

Shape

Description automatically generated with medium confidence

eq. 13

Injecting eq. 13 into eq. 10:

Shape

Description automatically generated with medium confidence

eq. 13, 14

Injecting eq. 13 into eq. 11:

Shape

Description automatically generated with medium confidence

eq. 15

Almost there! We now need to inject eq. 13 into the fourth equation, eq. 12. However, eq. 12 has ***bn-1***but eq. 13 holds until ***bn-2***. Good news, we can use eq. 14 to get rid of ***bn-1***then inject ***bn-2****.*

Shape

Description automatically generated with medium confidence

eq. 16

All right! To summarize we have in this order eq. 15, 13, 16:

Shape

Description automatically generated with medium confidence

eq. 15, 13, 16

We can write this system as a matrix multiplication and solve it. Hold on, we’re there!

Shape

Description automatically generated with medium confidence

eq. 17

As you can see, the first matrix has only 3 diagonals filled with values, others are zeros. This kind of matrix is called, reasonably enough, a *tridiagonal* matrix. Algorithms exist to solve this type of systems efficiently, such as [Thomas Algorithm](https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm) which runs linearly in time. For the sake of simplicity, we won’t bother optimizing further and we will simply use the built-in functions of Numpy in Python to solve the system.

However, we are still missing the ***bi***points. To find these we simply make use of eq. 13 which works out all the ***bi***up until ***bn-2***and then eq. 12 which gives the last term, ***bn-1***.

Shape

Description automatically generated with medium confidence

eq. 13, 12

We are done at last! Let’s see how we can program this using Python.