Introduction

Bezier curves are the most fundamental curves, used generally in computer graphics and image processing. These curves are mainly used in interpolation, approximation, curve fitting, and object representation. In this article, I will demonstrate, in a very simple and straightforward way, how one can construct these curves and make use of them.

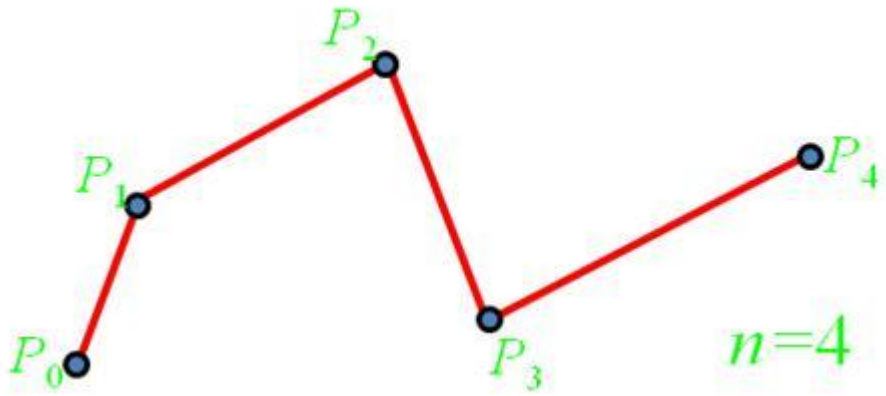
Background

Bezier curves are parametric curves which are pretty much customizable and smooth. They are well suited for many applications. They were named after Pierre Bézier, a French mathematician and engineer who developed this method of computer drawing in the late 1960s while working for the car manufacturer Renault. People say that at the same time the same development took place during the research of Ford. There is still a confusion about who found it first.

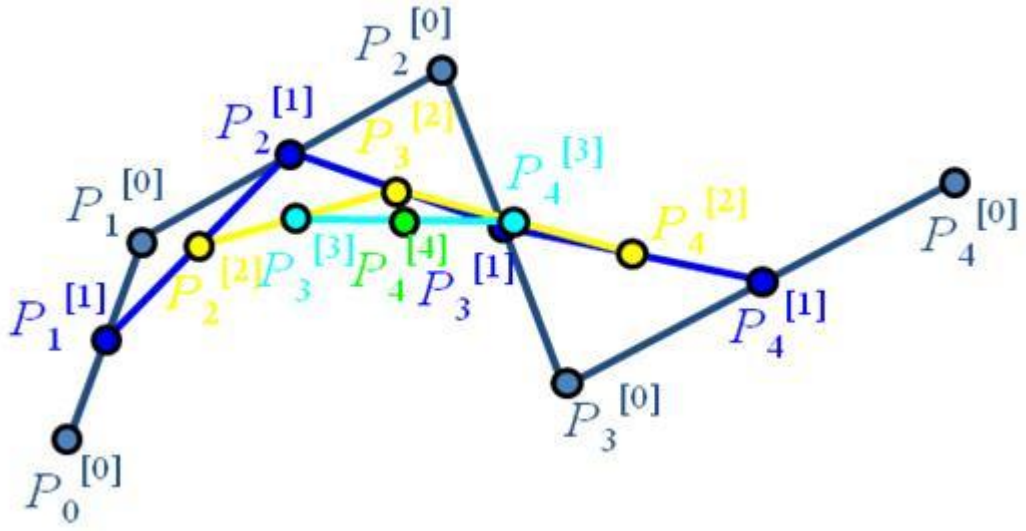
Because of my imaging background, my article will mainly focus on interpolation and curve fitting. In interpolation, what one would simply like to do is to find unknown points using known values. This way, a discrete case can be represented with a more continuous structure, and we can have a well defined curve for missing points. The curve is initialized with certain data points, and it tries to generate new ones that are approximating (or interpolating) the old values.

Constructive Bezier Curve Algorithm

Consider the n+1 points $P_0,\ldots,P_n$ and connect the points into a polyline we will denote hereafter as the control polygon.



Given points P_i, i = 0,...,n, our goal is to determine a curve g (t), for all values t Î [0,1]. The idea is demonstrated below:



Basic Algorithm

The objective here is to find points in the middle of two nearby points and iterate this until we have no more iterations. The new values of points will give us the curve. The famous Bezier equation is the exact formulation of this idea. Here is the algorithm:

Step 1: Select a value t Î [0,1]. This value remains constant for the rest of the steps.

Step 2: Set $P_i[0]$ (t) = P_i, for i = 0,...,n.

Step 3: For j= 0,...,n, set $P_i^{[j]}(t) = (1-t)P_{i-1}^{[j-1]}(t) + tP_i^{[j-1]}(t)$ for i = j,...,n.

Step 4: g (t) = $P_n[n]$ (t)

Special & General Cases

Now, I will give formulas for common, special cases that can be helpful in certain applications. The code of the article does not demonstrate any of them, but it uses the generalized formula. So, let me start with the generalized formula:

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} \mathbf{P}_i (1-t)^{n-i} t^i = \mathbf{P}_0 (1-t)^n + \binom{n}{1} \mathbf{P}_1 (1-t)^{n-1} t + \cdots + \mathbf{P}_n t^n \ , \ t \in [0,1].$$

For the sake of simplicity and convention used in this article and code, it is better to represent this formula as:

$$\gamma(t) = \sum_{i=0}^{n} P_i \frac{n!}{i!(n-i)!} (1-t)^{n-i} t^i ,$$

What this equation tells us is nothing but the formulation of the above algorithm (the mid-point iterations). It is very important in the sense that a whole algorithm could be summarized into a

formula and a straightforward implementation would yield correct results. Here, *n* denotes the number of points and *P* denotes the points themselves. The factorial coefficients of the points are simply called the Bernstein basis functions, because of the name of the founder.

Here are the special cases:

Linear Bezier:

$$\mathbf{B}(t) = \mathbf{P}_0 + (\mathbf{P}_1 - \mathbf{P}_0)t = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , t \in [0,1]$$

Quadratic Bezier:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2 \ , t \in [0,1].$$

Cubic Bezier:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3t(1-t)^2\mathbf{P}_1 + 3t^2(1-t)\mathbf{P}_2 + t^3\mathbf{P}_3 \ , t \in [0,1].$$

Understanding and Using the Code

This is the function, doing all the work. I think it is very short and very easy. Because we are dealing only with 2D curves, we have points in X and Y coordinates. The function simply calculates the Bezier points.

```
public void Bezier2D(double[] b, int cpts, double[] p)

{

   int npts = (b.Length) / 2;

   int icount, jcount;

   double step, t;


   // Calculate points on curve


   icount = 0;

   t = 0;

   step = (double)1.0 / (cpts - 1);


   for (int i1 = 0; i1 != cpts; i1++)
```

```
    {
        if ((1.0 - t) < 5e-6)
            t = 1.0;

        jcount = 0;
        p[icount] = 0.0;
        p[icount + 1] = 0.0;
        for (int i = 0; i != npts; i++)
        {
            double basis = Bernstein(npts - 1, i, t);
            p[icount] += basis * b[jcount];
            p[icount + 1] += basis * b[jcount + 1];
            jcount = jcount +2;
        }

        icount += 2;
        t += step;
    }
}
```