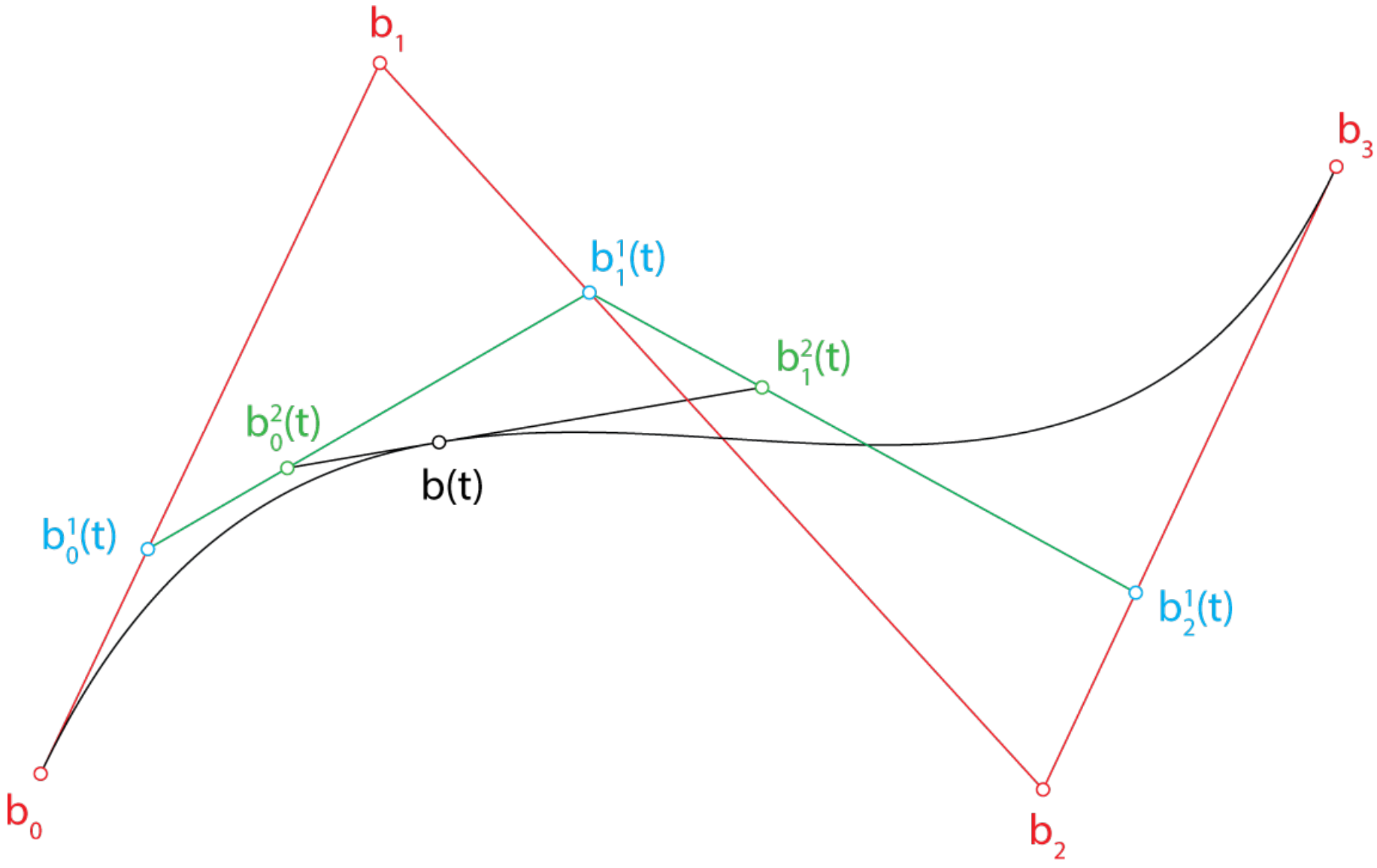
BÉZIER CURVES

DECONSTRUCTED



History

Before we get our hands dirty, a brief history of the Bezier Curve. There are 3 people whose contributions have made this curve what it is today.

1) Sergei Natanovich Bernstein

The Bernstein Polynomial was named after him. This served as the mathematical basis for the Bezier Curve. Unbeknownst to him when he introduced this, the polynomial restricted to the interval(0,1) became the Bezier curve 50 years later with the advent of Computer Graphics.

2) Paul De Casteljau

He was a French physicist and mathematician. In 1959, while working at Citroën, he developed an algorithm for evaluating calculations on a certain family of curves, which would later be formalized and popularized by engineer Pierre Bézier, and the curves called De Casteljau curve or Bézier curves.

De Casteljau's algorithm is widely used. Other methods, such as Horner's method and forward differencing, are faster for calculating single points but are less

robust. De Casteljau's algorithm is still very fast for subdividing a De Casteljau curve or Bézier curve into two curve segments at an arbitrary parametric location. A numerically stable way to evaluate polynomials in Bernstein form is de Casteljau's algorithm.

3) Pierre Bezier

He was a French engineer and one of the founders of the fields of solid, geometric and physical modelling as well as in the field of representing curves, especially in CAD/CAM systems. As an engineer at Renault, he became a leader in the transformation of design and manufacturing, through mathematics and computing tools, into computer-aided design and three-dimensional modelling.

Bézier patented and popularized the Bézier curves and Bézier surfaces that are now used in most computer-aided design and computer graphics systems. While Casteljau did work on the curve earlier than Pierre Bezier, Casteljau could not publish his findings for a few years.

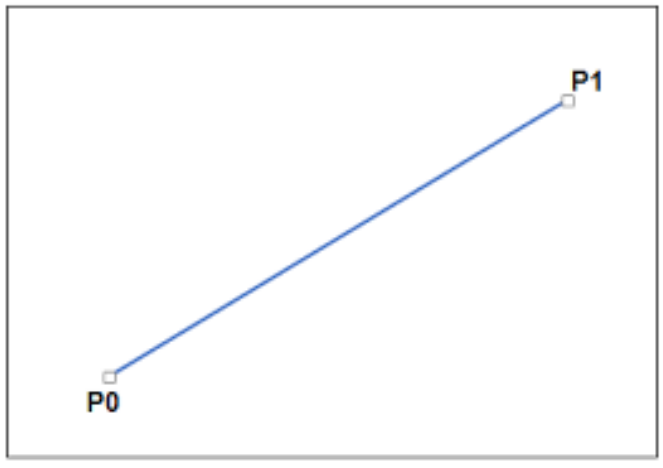
Just a heads up…. We are gonna be exploring the De Casteljau's algorithm.

WHAT IS IT?

A Bezier Curve is a parametric curve. It's used in general computer graphics, animation, fonts/typography. The bezier curve can be used to represent and transform complex curvilinear shapes.

A Bezier Curve has a start point, an end point, and maybe one or more control points depending on the type of Bezier Curve it is. The types of bezier curves are as follows:-

1)**Linear Bezier Curve** - Zero control points



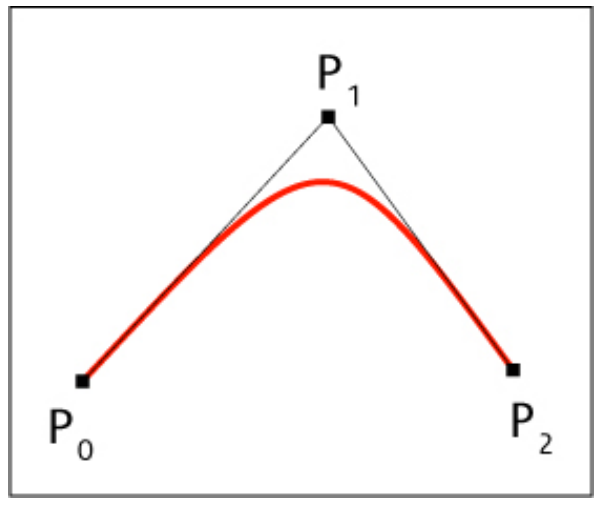
It's graphically represented as a straight line instead of a curve as it has no control points in the middle

In this example, the P0 is the starting point and P1 is the end point of the curve

2)**Quadratic Bezier Curve** - One control point

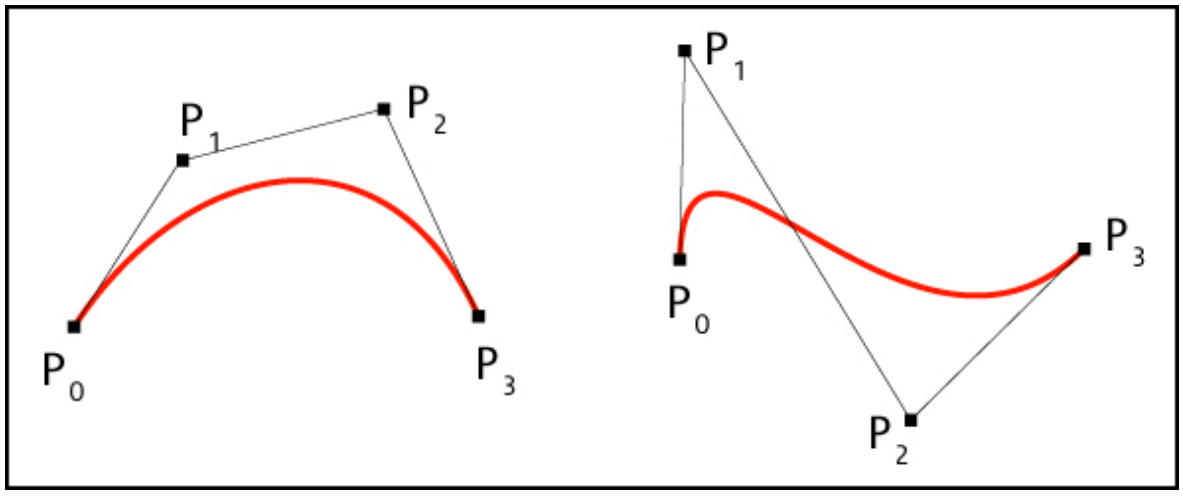
This curve is actually represented as a curve and it is pretty simple to create as we just have to define a single control point.

In this example, P0 is the starting point and P2 is the ending point of the curve and P1 is the control point that actually determines the shape of the curve



3)**Cubic Bezier Curve** - Two control points

This curve is a bit more complex as this is where we can start to see the difference between a normal curve and bezier curve.

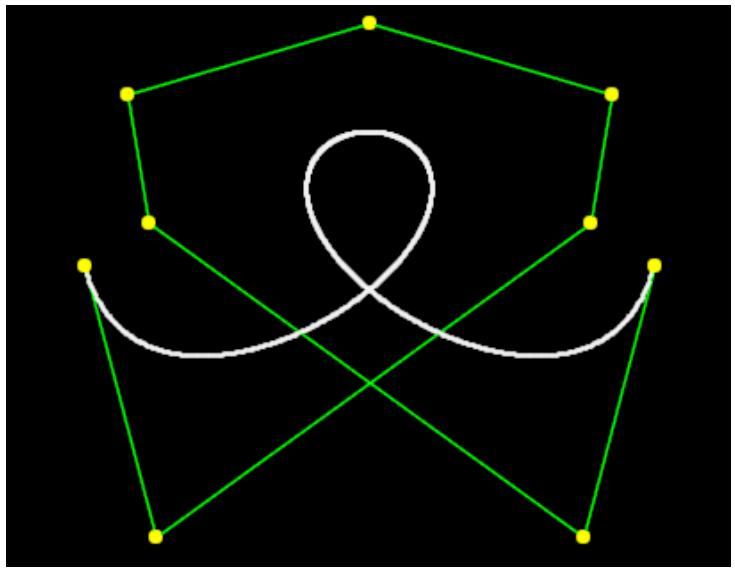


So in the example, P0 is the starting point and p3 is the ending point while the two control points are P1 and P2. While the example on the left can be considered a traditional curve the example on the right showcases the what can be achieved using the Bezier curve at the smallest scale.

4)**Higher order Bezier Curve** - Three or more control points
(I think this is just referred to as a Bezier curve)

Bezier Curves can be created with an arbitrary number of control points.

In this example you can there are 7 control points, and the curve is created based on those points. It is possible to make a more complex curve by increasing the number of control points.



*THE **MATH**:-*

First, let's take a look at some predefined Formulae that are widely used

In this instance, let's assume all the points are considered P_0 to P_n and the curve's coordinates are set by the parameter from the segment [0,1] with 0 as the starting of the curve and 1 as the end of the curve

Linear Bezier Curve(Zero Control point) -

$$P = (1-t)P_1 + tP_2$$

Quadratic Bezier Curve(Zero Control point) -

$$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^3P_4$$

Cubic Bezier Curve(Zero Control point) -

$$P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$$

All of the above formulae can be derived from De Casteljau's algorithm. We can cook up a different formula if the number of control points differ, but if we use a formula that was derived for a specific number of control points, we will not be able to use a random number of control points.

We will have to use De Casteljau's algorithm to create bezier curves with a degree of n whose value is arbitrary.

Parametric Functions

The Bezier Curve is a parametric function. A parametric function is a function where we give a single variable as input and get back multiple variables as outputs.

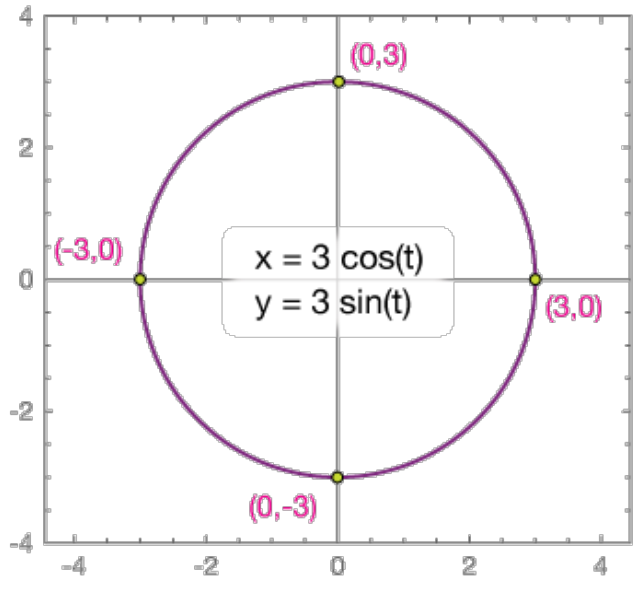
An explicit function may be something like this:

$$f(x) = sin(x)$$

Now an example of a Parametric function :

$$f(t) = \begin{cases} x = sin(t) \\ y = cos(t) \end{cases}$$

So as you can see in the Parametric function when we change the value of "t" the value of both the "x" and "y" variables change. With the input variable "t" we are able to acquire the values of both "x" and "y" variables which we will need to plot the curve.This specific function can be used to plot a curve



In certain cases, the Parametric functions will be able to other complex shapes by themselves as a single function. A Bezier Curve can also be one of the shapes.

De Casteljau's algorithm:

$$B(t) = \sum_{i=1}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i$$

(or)

$$B(t) = \sum_{i=1}^{n} {}^{n}C_i \, (1-t)^{n-i} t^i P_i$$

Alright, this the all you need to know to draw a Bezier curve of any degree. Trust me, no more math.

Lets dissect the equation to see what each part does

$$B(t)$$

This is the parametric function. The value of "t" ranges from 0 to 1.

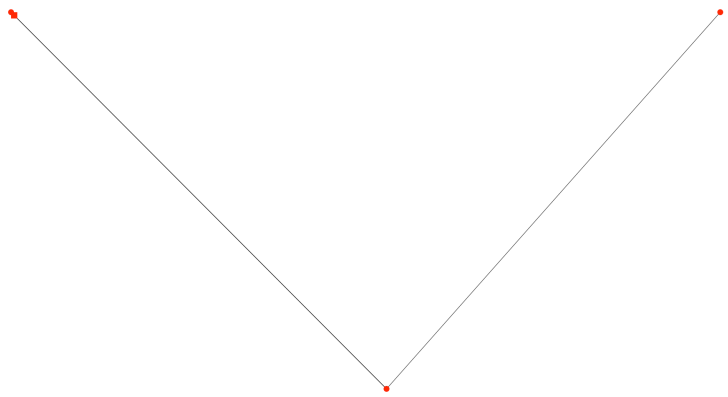
We plot the Bezier curve gradually from 0% to 100%.

When "t" value is 0 - we will get the "x" & "y" coordinates of the curve when it is at 0%, which should be the starting point of the curve

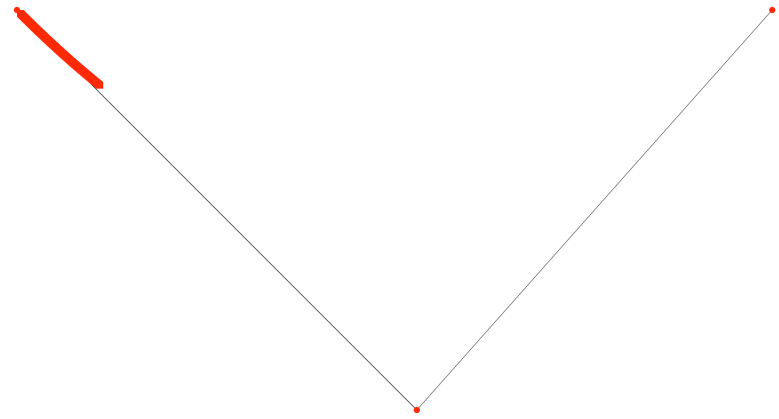
When "t" value is 1 - we will get the "x" & "y" coordinates of the curve when it is at 100%, which should be the ending point of the curve

So to get the "x" & "y" coordinates of the whole curve we should just keep incrementing the "t" value by 0.01 starting from 0 up until 1. This way we will get the "x" & "y" coordinates for each Percentage of the curve's length.

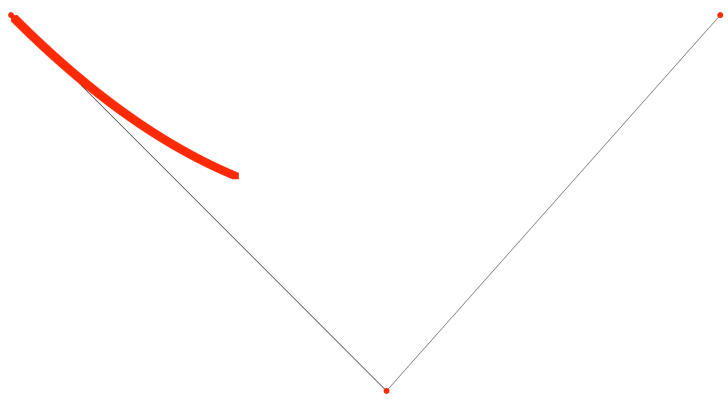
Lets look at a few examples with different "t" values:



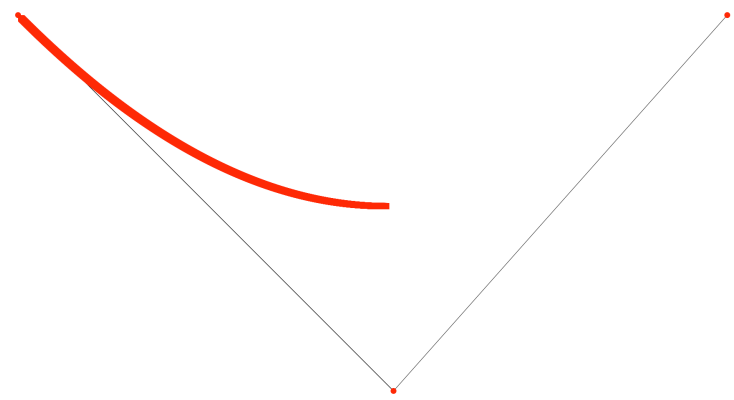
$t = 0$ (0%)



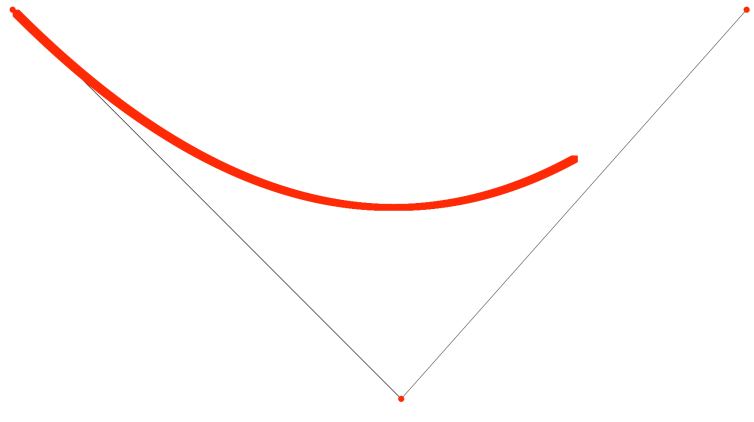
$t = 0.1$ (10%)



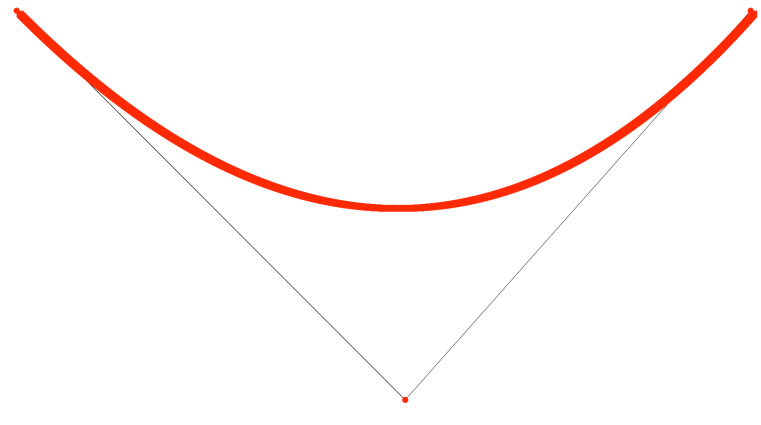
$t = 0.3$ (30%)



$t = 0.5$ (50%)



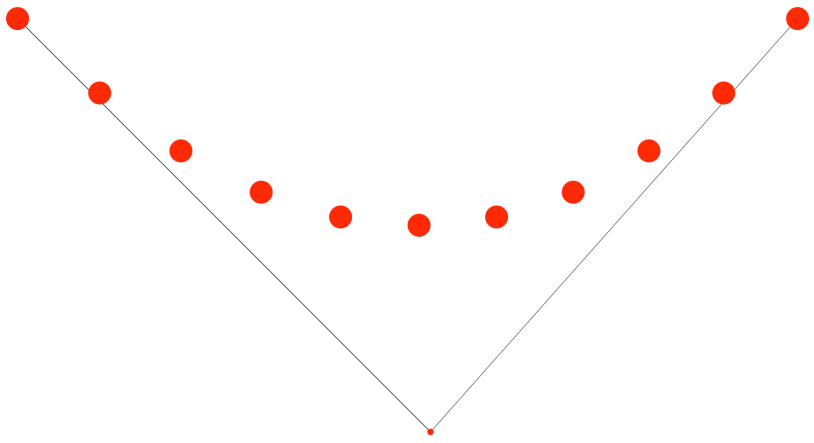
$t = 0.75$ (75%)



$t = 1$ (100%)

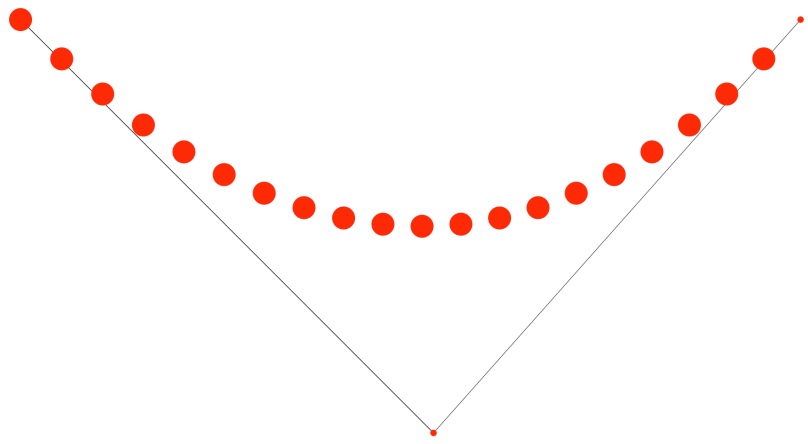
In all the examples above the "t" has been incrementing from 0 to its value to create the curve upto that point.

In the next example, we will see points in the curve when the "t" value gradually increases from 0 to 1 in **incrementing 0.1** in each iteration.



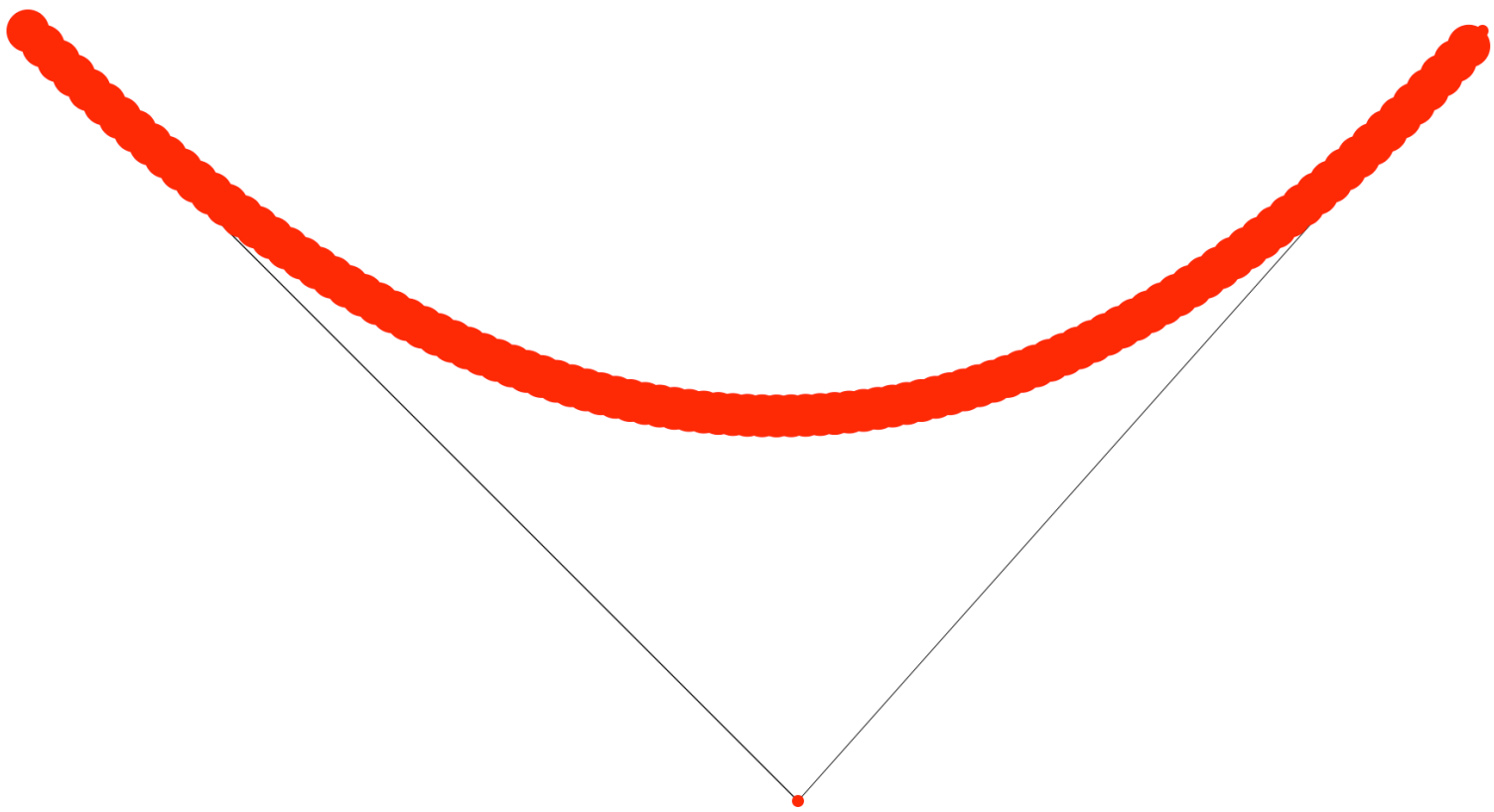
The red dots show the points in the Bezier Curve when the "t" value is 0, 0.1, 0.2, 0.3, ……., 0.9 and 1

In the next example, we will see points in the curve when the "t" value gradually increases from 0 to 1 in **incrementing 0.05** in each iteration.



The red dots show the points in the Bezier Curve when the "t" value is 0, 0.05, 0.1, 0.15, ……., 0.95

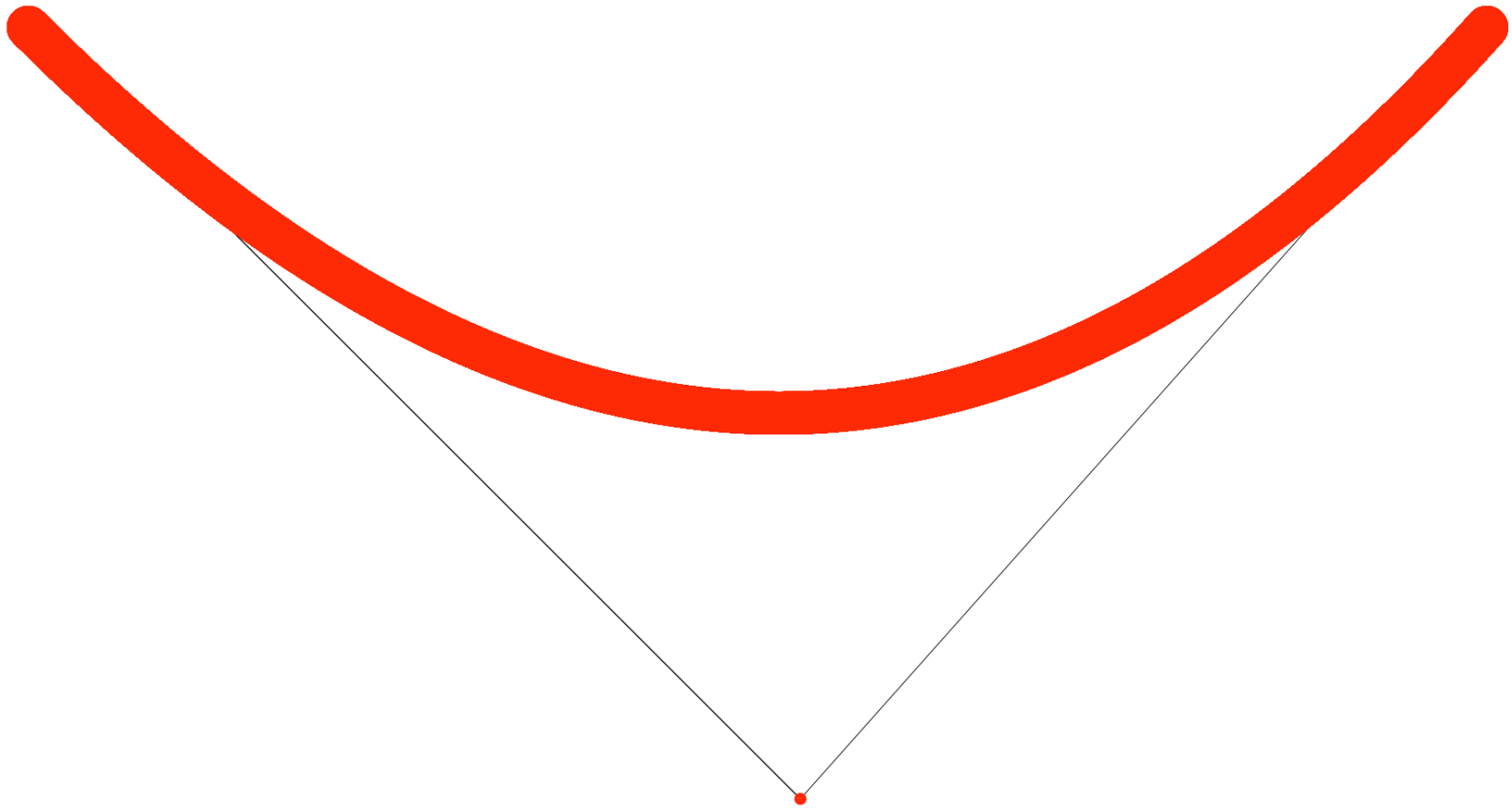
In the next example, we will see points in the curve when the "t" value gradually increases from 0 to 1 in **incrementing 0.01** in each iteration. We can get the coordinates for every single percentage of the length of the Bezier curve.



The red dots show the points in the Bezier Curve when the "t" value

is 0, 0.01, 0.02, ….., 1. If you notice closely, the each circle is still visible. If we reduce the value it increments in each iteration, the curve will be more smooth.

In the next example, we will see points in the curve when the "t" value gradually increases from 0 to 1 in **incrementing 0.001** in each iteration.



As you can see we made a Bezier Curve by just plotting a bunch of points in its path. The bigger the curve and more smoother you want the curve to be…. Reduce the incrementing value accordingly.

$$^{n}C_{i}\,(or)\binom{n}{i}$$

This is what is known as a binomial coefficient. The "C" in $^{n}C_{i}$ is from Permutations and Combinations. It should be read as "n" Choose "i". This is pretty interesting actually, look into it if you have nothing else to do. There is a simple formula for this.

$$^{n}C_{i} = \frac{n!}{i!\,(n-i)!}$$

$$\sum_{i=0}^{n}$$

This is known as the summation as you probably already know. Here “i” is the lower limit and “n” is the upper limit. Now in the equation following the summation, we will have to recursively increase the value of “i” till it reaches the value of the variable “n” and add it to the answer of the previous iteration.

$$\sum_{i=1}^{n} {}^{n}C_i\,(1-t)^{n-i}t^{i}P_i$$

So in this equation “n” resembles the number of control points. So we increment the value of “i” and substitute it into the equation until “i” is equal to “n”.

For example let’s assume we have to calculate B(t) for a cubic bezier, we will have to go through following steps

$$B(t) = \sum_{i=1}^{n} {}^{n}C_i\,(1-t)^{n-i}t^{i}P_i$$

As it is cubic bezier “n” =3

Loop 1 - $(1-t)^3P_0$

Loop 2 - $3(1-t)^2tP_1$

Loop 3 - $3(1-t)t^2P_2$

Loop 4 - t^3P_3

Now we have to add everything from each iteration together :

$$B(t) = (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3$$

We have arrived at the general formula for Cubic Bezier. So if you need to figure out the formula for a different value of “n”, you can use the above method.

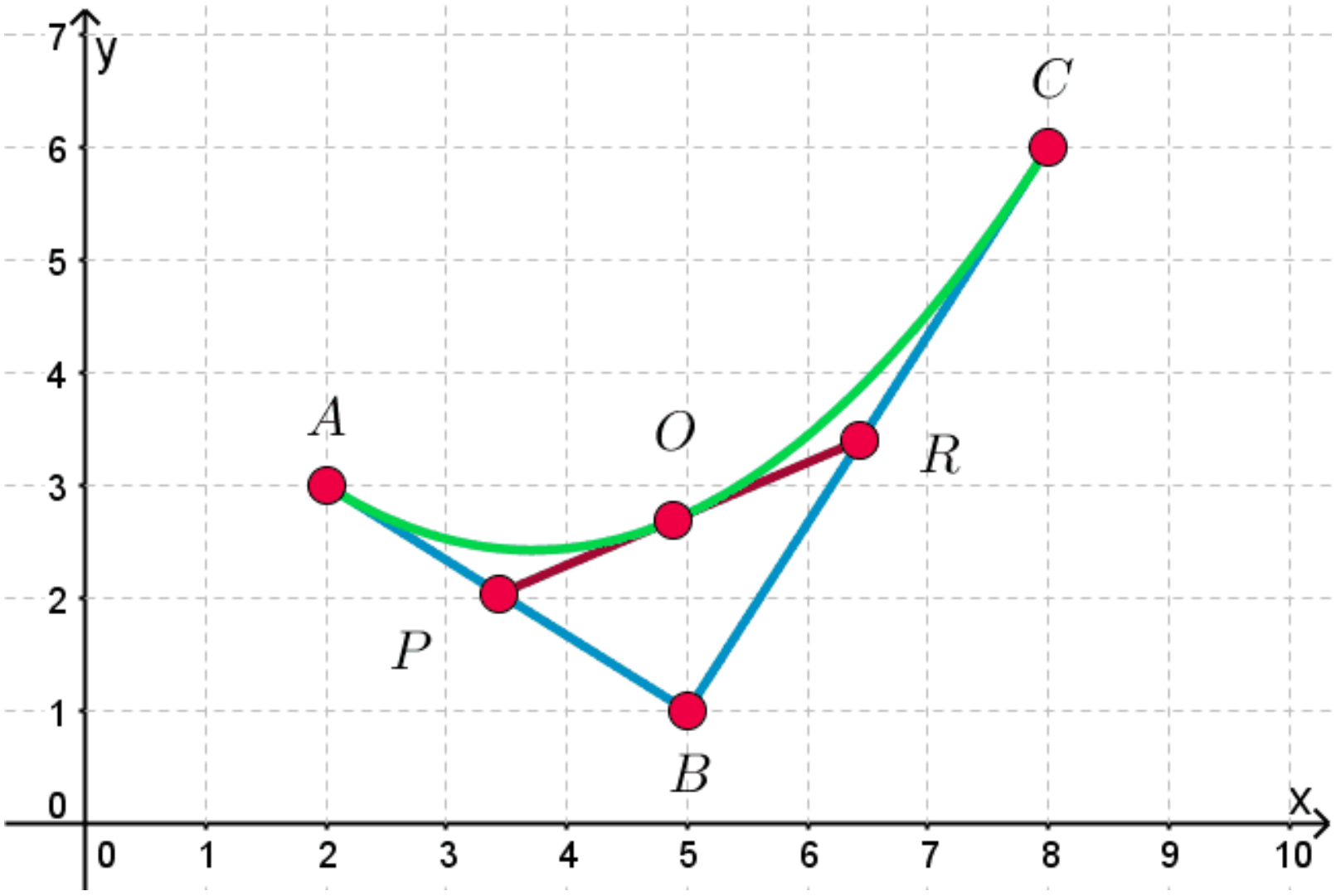
In this, if we have “t” = 0, then B(0) = P_0 as the start of the curve is P_0

If “t” = 1, then B(1) = P_n in the case of cubic bezier B(1) = P_3 as that is the end of the curve.

So to compute the coordinates for the middle of the curve we can make “t” = 0.5 and so on.

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

Using this formula and incrementing the value of “t” gradually, we will be able to draw the cubic bezier.



In the above cubic bezier curve, the “t” value is 0.5.

Deconstructing the ***CODE***

Check out this Github Repo for the complete Html file

https://github.com/fobox/BezierCurve/

Now we are going to look into the underlying logic and try to understand it so that you gain the ability to implement this elsewhere.

```
//Calculating the factorial
    function factorial(x) {
        if(x==0) {return 1;}
        return x * factorial(x-1);
    }


//Calculating the Binomial Coefficient
function combination(n,i){
   return (factorial(n)/(factorial(i)*factorial(n-i)));
}
```

This piece of code is used to calculate the Binomial Coefficient.

$$ {}^{n}C_{i} = \frac{n!}{i!\,(n-i)!} $$

The first function is to calculate the factorial alone and the second function uses the factorial function to actually calculate the binomial coefficient using the general formula.

```
  var controlPoints =[];

function bezierCurve(t) {

        var xCoordinate=0;
        var yCoordinate =0;
        var n=controlPoints.length-1;

           for(i=0;i<=n;i++){

           xCoordinate = xCoordinate +
   (combination(n,i)* Math.pow((1-t),(n-i))*Math.pow(t,i)*controlPoints[i].x);

           yCoordinate = yCoordinate +
    (combination(n,i)* Math.pow((1-t),(n-i))*Math.pow(t,i)*controlPoints[i].y);

           }

            return [xCoordinate, yCoordinate];
    }
```

This is the code that calculates the Bezier Curve. As we need both “x” and “y” coordinates plot parts of the graph we add calculate the “x” coordinate separately and the “y” coordinate separately.

$$B(t) \,.\, x = \sum_{i=1}^{n} {}^{n}C_i \,(1-t)^{n-i} t^i (P_i \,.\, x)$$

And

$$B(t) \,.\, y = \sum_{i=1}^{n} {}^{n}C_i \,(1-t)^{n-i} t^i (P_i \,.\, y)$$

In the code

```
yCoordinate = yCoordinate +
 (combination(n,i)* Math.pow((1-t),(n-i))*Math.pow(t,i)*controlPoints[i].y);
```

1) `combination(n,i)` = $^{n}C_{i}$

2) `Math.pow((1-t),(n-i))` = $(1-t)^{n-i}$

3) `Math.pow(t,i)` = t^{i}

4) `controlPoints[i].y` = $P_{i}.y$

5) `controlPoints[i].x` = $P_{i}.x$

6) In each iteration till "i" reaches "n", the equation's solution is added to the previous iteration's solution (`yCoordinate = yCoordinate + [equation]`) = $\sum_{i=0}^{n}$

```
//This is where we are drawing the Bezier curve
       var drawxCoordinate, drawyCoordinate;
      for (t=0;t<=1;t=t+0.001) {
          [drawxCoordinate, drawyCoordinate] = bezierCurve(t);
          ctx.fillRect(drawxCoordinate, drawyCoordinate,5,5);
      }} );
```

So, finally we are at B(t) where we increase the value of "t" gradually from 0 to 1 to form the whole curve. We increment the value by 0.001 so the curve will look very smooth. We can use the line to and make the curve by connecting a bunch of lines….. I would personally recommend you to use the lines instead of rectangles. For experimental purposes I have been using squares and also because squares render faster in the canvas compared to circles.

https://jsperf.com/canvas-rect-vs-circle

Well technically it could be a better idea to draw a line between the points rather than drawing a square for each point … but I am using squares cause I think that it is a bit more easier to understand when shown as separate points.

You can try out the code in the GitHub repo to get a clearer understanding of its implementation. You can also play around with the random Bezier Curve generator in the same repo.

Let me know if you find any bugs and please feel free to ask me if you have any other related issues.

Here are some other places online you can learn more about Bezier Curves :

https://en.wikipedia.org/wiki/Bernstein_polynomial

https://en.wikipedia.org/wiki/Paul_de_Casteljau

https://en.wikipedia.org/wiki/B%C3%A9zier_curve

https://en.wikipedia.org/wiki/Pierre_Bézier

https://en.wikipedia.org/wiki/Parametric_equation

https://www.researchgate.net/post/Who_first_defined_the_so-called_Bezier_curves

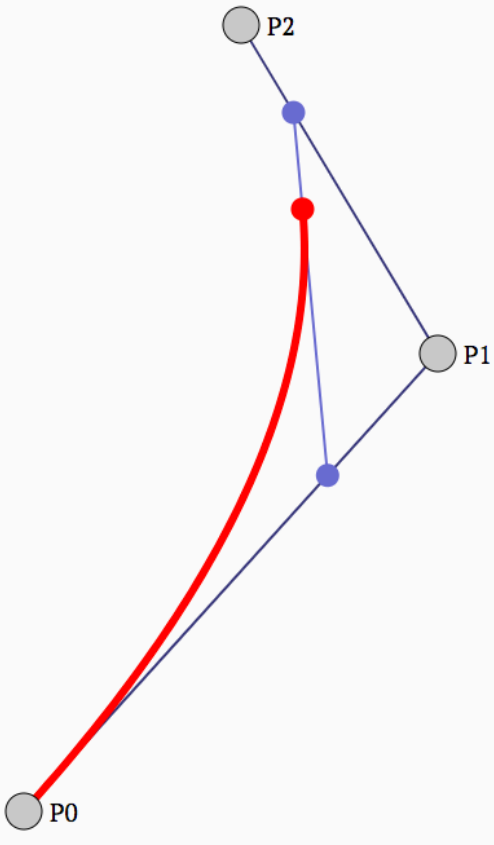
https://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm

https://pomax.github.io/bezierinfo/

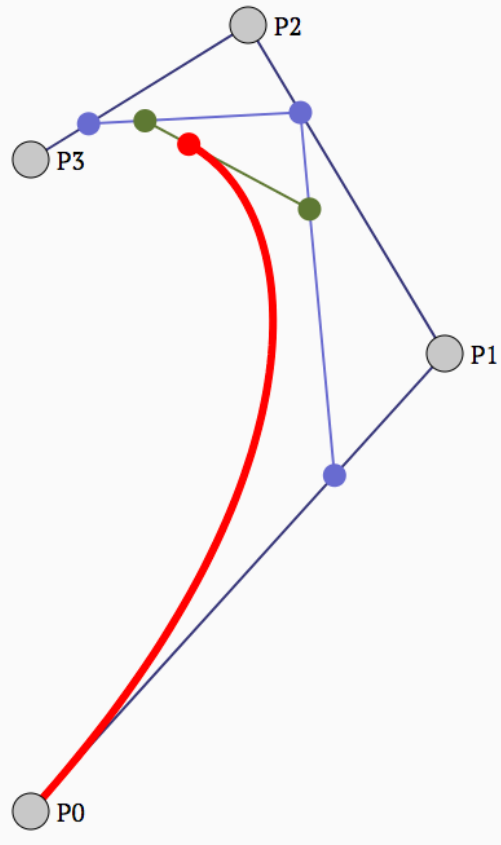
https://buildingvts.com/mathematical-intuition-behind-bezier-curves-2ea4e9645681

https://medium.freecodecamp.org/nerding-out-with-bezier-curves-6e3c0bc48e2f

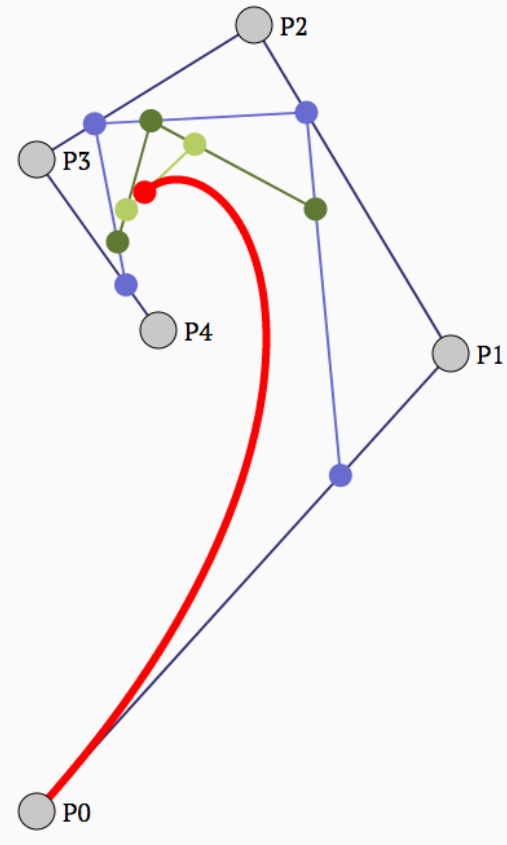
https://javascript.info/bezier-curve



t=0.73



t=0.73



t=0.73

If this makes sense to you now….. I guess you are good to go.

Hope you found this somewhat useful.

Until next time.

MAY THE FORCE BE WITH YOU