Jessica Garrett
Numerical Analysis Project 2
04/29/2014

Bezier Curves and Data Fitting

Bezier curves are a playful application of computer graphics, such as font design. They are defined first by four point in Cartesian coordinates, (x_1, y_1) (x_2, y_2) (x_3, y_3) (x_4, y_4) where (x_1, y_1) and (x_4, y_4) are endpoints and the middle two are "control points". Our curve "leaves" (x_1, y_1) in the direction of the first control point (x_2, y_2), i.e, along the tangent line defined by $(x_2 - x_1, y_2 - y_1)$. Likewise, (x_3, y_3) (x_4, y_4) express the later part of the curve. At value t, where $t \in [0,1]$ our coordinates are defined parametrically as

$$x(t) = x_1 + b_x + c_x t^2 + d_x t^3$$

$$y(t) = y_1 + b_y + c_y t^2 + d_y t^3$$

and the coefficients represented as

$$b_x = 3(x_2 - x_1)$$

$$c_x = 3(x_3 - x_2) - b_x$$

$$d_x = x_4 - x_1 - b_x - c_x$$

and similarly for b_y, c_y and d_y. After computing coefficients, we consider x(t) and y(t) at various values of t, say t = [0:0.1:1], and plot the Bezier curve along projected points. If t is defined as stated, we will have 101 Cartesian points to consider.

3.1 Fun with Bezier Curves

1. First we must write a function, get_coeffs, that accepts two end points and two control points and returns a coefficient vector of size (1,2).

```
function [ xt yt ] = get_coeffs( ctrl1, ctrl2, edpt1, edpt2 )
%Takes four points and computes a x(t) and y(t) which defines a Bezier curve.
%Inputs: ctrl1, ctrl2 = control points; (x2,y2) and (x3,y3)
%        edpt1,edpt2 = endpointsl (x1,y1) and (x4,y4)
%%Outputs: xt, yt = coefficients needed to define Bezier curve.
%
%Varibles bx,cx,dx,by,cy,dy are defined as...
clc; close;
bx = 3*(ctrl1(1)- edpt1(1));
cx = 3*(ctrl2(1)- ctrl1(1));
dx = edpt2(1) - edpt1(1) - bx -cx;
by = 3*(ctrl1(2)- edpt1(2));
cy = 3*(ctrl2(2)- ctrl1(2));
```

```
dy = edpt2(2) - edpt1(2) - by - cy;

%setup coeff as vector
xt = [edpt1(1), bx, cx, dx];
yt = [edpt1(2), by, cy, dy];
end
```

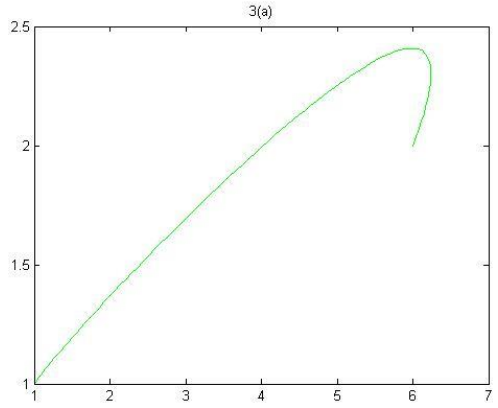
2. Second, we must actually points along the curve using the function values x(t) and y(t) via the function make_bezier

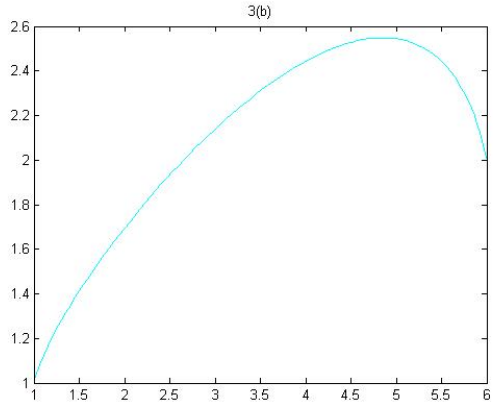
```
function [ xtval ytval ] = make_bezier( xt, yt, t )
%Returns a (x,y) pair corresponding to the position on Bezier curve at
%value t; where t is between (0,1)
%Inputs: xy, yt = coefficient vectors both of size 4?
%        t = user defined parameter taking on values (0,1)
%Outputs: pair = point (x,y) on Cartesian coordinates representing position
%at value t.
%xtval = numerical value(s) corresponding to x(t) at specified t.
%ytval = numerical value(s) corresponding to y(t) at specified t.

t = [0:.01:1]; %define t as points between 0 and 1
new_xt = fliplr(xt); %descending order for use of polyval function
new_yt = fliplr(yt);
xtval = polyval(new_xt,t);
ytval = polyval(new_yt,t);
end
```

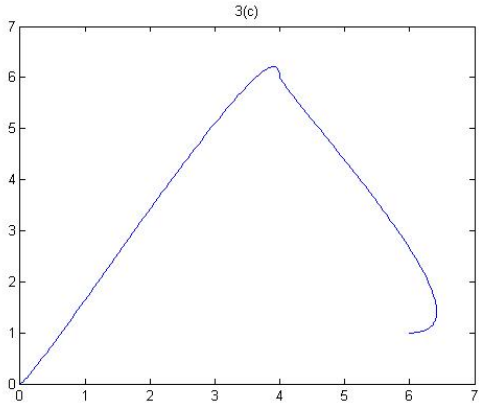
3. The make_bezier function simply returns an array of x and y values of equal quantities. To actually create a Bezier curve we must plot our findings. Let's test our functions by graphing the cubic curve associated with endpoints and control points 3(a) – 3(c). Below is the plotting code as well as the results…

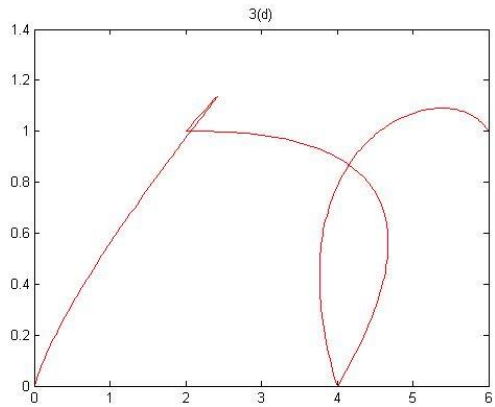
```
%Script calls functions get_coeffs and make_bezier and plots Bezier
%polynomials with t values from 0 to 1 in 0.01 intervals.
clear; clc;

%(a) points: (1,1) (6,2); control points: (1.5, 1.25) (7,3)
a1=[1.5 1.25]; b1=[7 3]; c1=[1 1]; d1=[6 2];
[xt1 yt1] = get_coeffs(a1,b1,c1,d1);
[xtval1 ytval1 ] = make_bezier( xt1, yt1 );
subplot(2,2,1);
plot(xtval1, ytval1,'g');
title('3(a)')

%(b) points: (1,1) (6,2); control points: (1.25, 1.5) (5,3)
a2=[1.25 1.5]; b2=[5 3]; c2=[1 1]; d2=[6 2];
[xt2 yt2] = get_coeffs(a2,b2,c2,d2);
[xtval2 ytval2 ] = make_bezier( xt2, yt2 );
subplot(2,2,2);
plot(xtval2, ytval2,'c');
title('3(b)')

%(c) points: (0,0) (4,6); control points: (.5, .5) (3.5,7) (first part)
a3a=[.5 .5]; b3a=[3.5 7]; c3a=[0 0]; d3a=[4 6];
```

```
[xt3a yt3a] = get_coeffs(a3a,b3a,c3a,d3a);
[xtval3a ytval3a ] = make_bezier( xt3a, yt3a );

%points: (4,6) (6,1); control points; (4.5,5) (7,2)  (second part)
a3b=[4.5 5]; b3b=[7 2]; c3b=[4 6]; d3b=[6 1];
[xt3b yt3b] = get_coeffs(a3b,b3b,c3b,d3b);
[xtval3b ytval3b ] = make_bezier( xt3b, yt3b );
subplot(2,2,3);
plot(xtval3a, ytval3a,'b', xtval3b, ytval3b, 'b'); %plot both
title('3(c)')

%(d) points: (0,0)(2,1); control points: (0.5,0.5) (3,1) (first part)
a4a=[.5 .5]; b4a=[3 1]; c4a=[0 0]; d4a=[2 1];
[xt4a yt4a] = get_coeffs(a4a,b4a,c4a,d4a);
[xtval4a ytval4a ] = make_bezier( xt4a, yt4a );
%points: (2,1) (4,0); control points; (3,1) (5,1)  (second part)
a4b=[3 1]; b4b=[5 1]; c4b=[2 1]; d4b=[4 0];
[xt4b yt4b] = get_coeffs(a4b,b4b,c4b,d4b);
[xtval4b ytval4b ] = make_bezier( xt4b, yt4b );
%points: (4,0) (6,1); control points; (3,1) (6.5,0.25)  (third part)
a4c=[3 1]; b4c=[6.5 0.25]; c4c=[4 0]; d4c=[6 1];
[xt4c yt4c] = get_coeffs(a4c,b4c,c4c,d4c);
[xtval4c ytval4c ] = make_bezier( xt4c, yt4c );
subplot(2,2,4);
plot(xtval4a, ytval4a,'r', xtval4b, ytval4b, 'r', xtval4c,ytval4c, 'r');
%plot all three
title('3(d)')
```







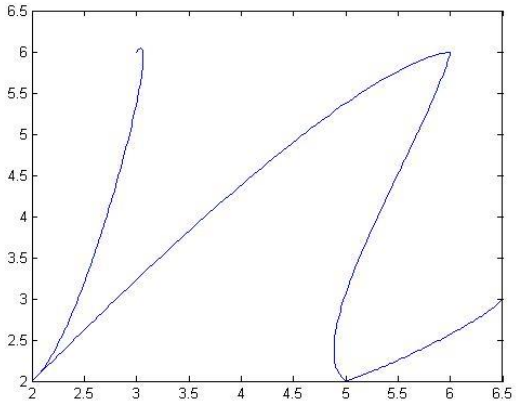


4. Here, we can see Bezier curves in action. In parts c and d we can see that the curve is really defined as multiple Bezier curves with appropriate starting and ending points for each segment. How can we considering just a general matrix of endpoints/control points where each row is one segment? Functions, bezier_coeffs and Bezier_poly demonstrate just this…

```
function [ xt yt numrow] = bezier_coeffs( p )
%Takes a matrix p of dimensions (m x 8), where each row is a x,y end/control
%point and returns a x(t) value and a y(t) value needed to define curve. Also
%returns number of rows in matrix p.

numrow = size(p,1) %returns number of rows

%Variables bx,cx,dx,by,cy,dy are defined as...
clc; close;

for k=1:numrow
    bx = 3*(p(k,3) - p(k,1));
    cx = 3*(p(k,5) - p(k,3)) - bx;
    dx = p(k,7) - p(k,1) - bx - cx;

    by = 3*(p(k,4) - p(k,2));
    cy = 3*(p(k,6) - p(k,4)) - by;
    dy = p(k,8) - p(k,2) - by - cy;

    %setup coefficients as "cell array vector"
    xt{k} = [p(k,1), bx, cx, dx];  %each "element" in xt and xt is a 1x4
    yt{k} = [p(k,2), by, cy, dy];
end
end

function [ xval yval ] = bezier_poly(xt,yt,n)
%Takes coefficient matrix for bezier curve and computs a numerical x and y
%value.  xt, for example is in the form where each element is a cell array,
%therefore consider the "cell2mat" built-in function. n=# of rows in
%original matrix p, which all three inputs were derived from...

xval = zeros(n,101); %pe-establish arrays of values
yval = zeros(n,101);
t = [0:.01:1];

for j = 1:n                          %consider x values
    xt_new = cell2mat(xt(j));
    xt_newer = fliplr(xt_new);
    xval(j,1:101) = polyval(xt_newer,t);

    yt_new = cell2mat(yt(j));        %consider y values
    yt_newer = fliplr(yt_new);
    yval(j,1:101) = polyval(yt_newer,t);
end
end
```

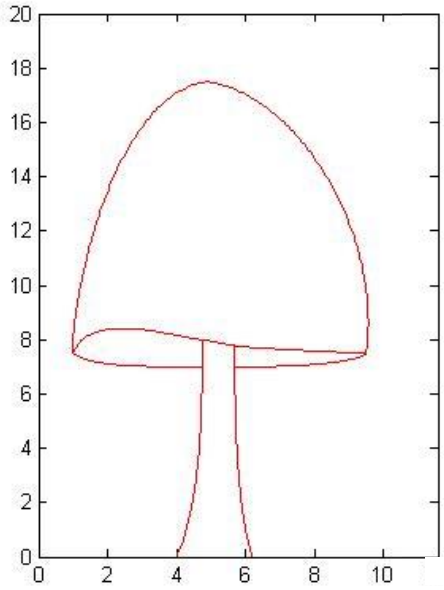
To test our general all-purpose functions let's consider the matrix p and plot the associated Beizer curves. This four part curve traces the letter "n".

```
p = [3 6 3.3 6.5 2.5 2.5 2 2;
     2 2 2.8 3 5 5.8 6 6;
     6 6 5.8 5 4.5 2.5 5 2;
     5 2 5.5 2.2 6.4 2.8 6.5 3]
[x y n] = bezier_coeffs(p);
[xval yval] = bezier_poly(x,y,n);

for k = 1:n
    plot(xval(k,1:101),yval(k,1:101));
    axis([2 6.5 2 6.5])
    hold on
end
```



5. In theory, any two dimensional sketch can be represented by a matrix of size (m x 8) and then expressed as Bezier curves. Let's look now at the matrix m and its associated curve... (Note- the asymmetry was intentional)

```
m = [4 0 4.75 2 4.75 6 4.75 8;
     4.75 7 3 7 1.5 7 1 7.5
     1 7.5 1.5 9.2 4.2 8 4.75 8
     4.75 8 4.9 8 5.5 7.8 5.7 7.8
     5.7 7.8 5.6 7.7 9.4 7.5 9.5 7.5
     9.5 7.5 9 7 6.5 7 5.7 7
     9.5 7.5 10 12 7.5 17 4.9 17.5
     4.9 17.5 2 17 0.9 9 1 7.5
     5.7 7.8 5.68 6 5.7 2 6.2 0]

[x y n] = bezier_coeffs(m);
[xval yval] = bezier_poly(x,y,n);

for k = 1:n
    plot(xval(k,1:101),yval(k,1:101),'r');
    axis([0 20 0 20])
    hold on
end
```



3.2 Fitting Bezier Curves

3.2.1 Derivative (gradient) approximation

1) Let's consider the 3rd degree Taylor Series Approximation for following functions:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(c_1) \qquad x < c_1 < (x+h)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(c_2) \qquad (x-h) < c_2 < x$$

Subtract $f(x+h) - f(x-h)$ and solve for $f'(x)$…

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{12}f'''(c_1) - \frac{h^2}{12}f'''(c_2)$$

We can see the first term is a slight variation of the definition of the first derivative, where, h is "small" $f'(x) \approx \frac{f(x+h)-f(x)}{h}$. Therefore, the last two terms are sources for error. However, the Generalized Intermediate Value Theorem suggests that the last two terms can be combined, where our now arbitrary constant $c \in (x-h, x+h)$. Our finalized formula for $f'(x)$ is known as the Three-point centered-difference formula. Since the remainder term is of degree 2, $O(h^2)$, the Three-point centered-difference formula is an order 2 approximation. When the error term is omitted from this formula, we are of course considering just an approximation of the derivative.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(c)$$

2) Fixing h to be 0.001 consider the function dt that returns the approximate derivative at x_0 given a function handle f and a numerical scalar x_0

```
function [ df_approx ] = df(f,x0)
%Comutes approximated derivative using 3-point central difference formula

%Inputs: f = user defined function handle
%        x0 = user defined scalar
format long
h = 0.001;
df_approx = (f(x0+h) - f(x0-h))/(2*h);
end
```

Defining our function as $f = @(x)\ 2 * x.\hat{}2$ and $x_0 = 3$, function df returns the value 11.999999999998678. The true derivative at $x_0 = 3$ is $f'(3) = 4(3) = 12$. The forward relative error is just -1.101341240428155e-13. Again, let's now consider the function handle as $g = @(x)\ \exp(3 * x)$. We see that the approximate derivative using the Three-point centered-difference formula is 2.430928824662124e+04. The true derivative is $g'(3) = 3e^{3*3}$= 2.430925178272615e+04. Again, our computed results are also near identical to the true derivative. The relative error for our second test is 1.500000716227131e-06.

3.2.3 Gradient Descent

In the previous section, we used the three point difference formula, omitting the last term, to approximate the derivative. If we were to consider the error term in our formula, how might we attempt to minimize this error function, as of course this would yield a better approximation. Gradient descent is a technique we may consider and is defined iteratively by

$$x_{i+1} = x_i - \gamma f'(x_i)$$

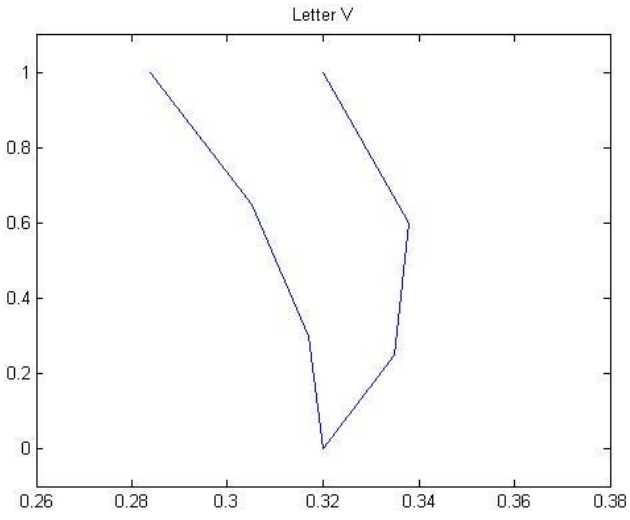
Using our df function to derive $f'(x_i)$ let's consider the following function, descent, to compute a value x that minimizes an error function.

```
function [ g ] = descent(f,x0)
%Gradient descent via iterative methods. Takes function handle f and initial
```

```
%value to first approximate the derivative via three point centered
%difference function, df.

format long
fp = df(f,x0);
eps = 0.1;
x(1) = x0;
iter = 0;
 for i = 1:501;
     fp = df(f,x(i));
     x(i+1) = x(i) - (eps*fp);
     iter = iter + 1;        %define counter
        if abs (x(i+1)-x(i)) < 0.0001  %stopping conditions
            break
        end
  end
g = x(i+1);
disp(iter);
end
```

1) The function, descent, fixes γ to be 0.1 and uses the stopping criteria of 500 iterations or when the difference between the current x_i and the previous is less than 0.0001. Let's this function with the error function $E(x) = (x-2)^2$ using the initial value of $x_0 = 4$
2) The gradient at the initial point $x_0 = 4$ is $x = 4 - (0.1 * 4.000000000000448)$ = 3.599999999999955. Evaluating the error function at this value yields 2.559999999999855, which is quite large considering we want a value significantly close to zero.
3) Of course the true solution should be $x = 2$ as $E'(x) = 2(x-2) = 0$ when $x = 2$.
4) Executing $descent(f, 4)$ yields the computed x to be 2.000332306998947 with 39 iterations. The approximated solution is accurate within 3 decimal places. Redefining our stopping criteria with tighter tolerance would produce better results. For example changing The condition of $|x_{i+1} - x_i| < 10^{-15}$ would yield a result within 13 decimals places in 142 iterations.

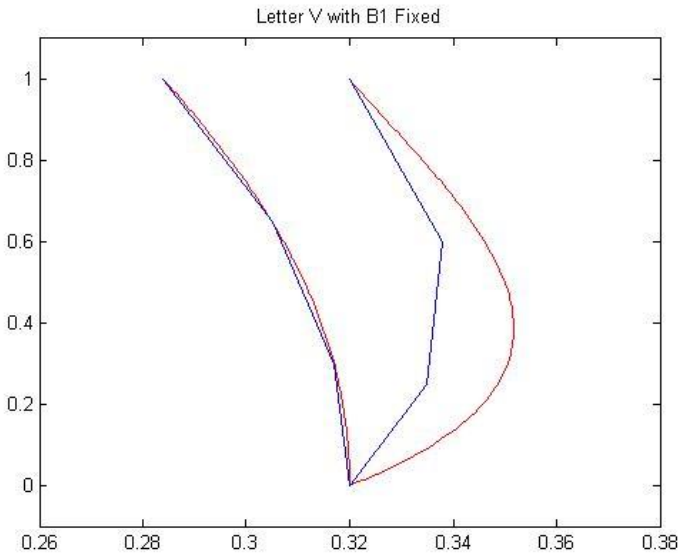
3.2.3 Fitting to data



Consider the following 7 points (0.284,1.0), (0.305, 0.65), (0.317,0.3), (0.32, 0.0), (0.335, 0.25), (0.338, 0.6), and (0.32, 1.0). The plot to the right demonstrates an approximate shape. At a glance, we can see than the figure closely resembles the letter "v". With the knowledge of Bezier splines we can modify this figure, or "fit the data" to resemble an even more recognizable letter "v". If one Bezier curve$B_1(t)$, fitting the left side of the letter, was given, could we complete the remaining right,$B_1(t)$? Let's look at the following matrix$B$, where $B_1$is $B(1,:)$and B_2 is $B(2,:)$and is set up as endpoint/control point as we saw in previous sections.

```
B = [0.284 1.0 0.305 0.7 0.32 0.4 0.32 0.0;
     0.32 0.0 0.38 0.3 0.34 0.7 0.32 1.0]
```

1) We can see that the left spline B_1 fits the data points very well. However, B_2, the right portion, is hyper-extended. Notably, the curve leaving the first endpoint (0.32, 0.0) is being "pulled" too far in the x direction by the control point (0.38, 0.3). In order to fix our graphical error, let's look at the Matlab function make_B2 that takes the first control point x_2 as an input variable and, coupled with similar code from get_coeffs, to return a new form of B_2. All other values remain unchanged.



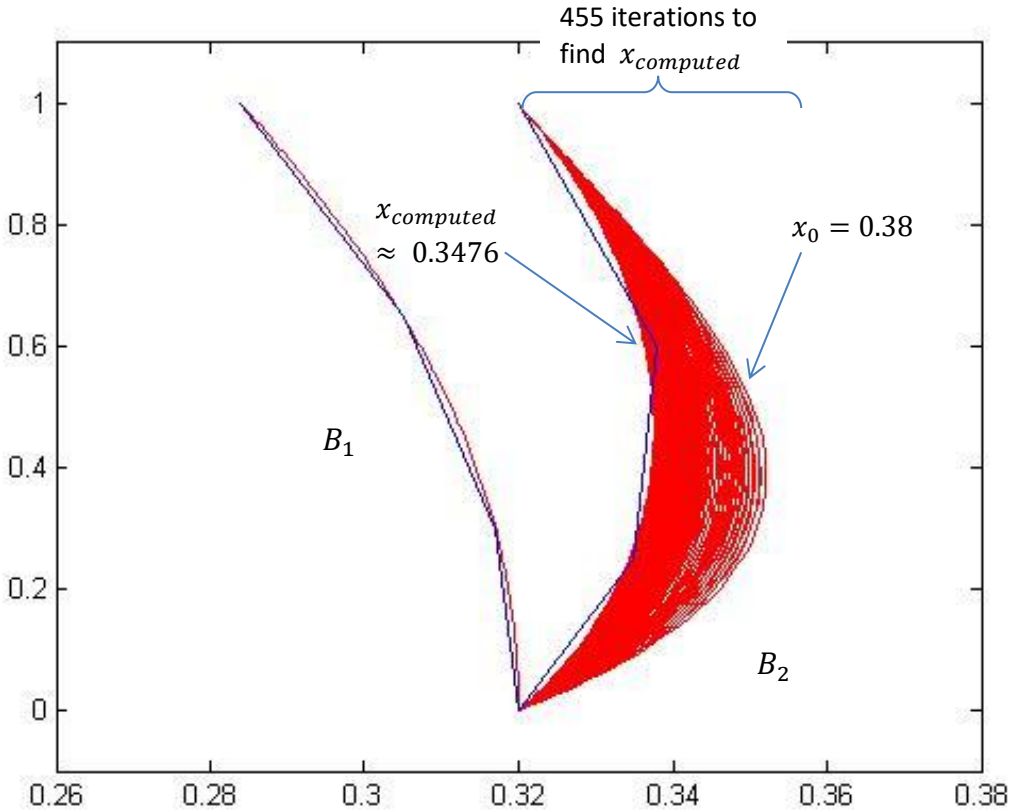
2)
```
function [ B2x_new B2y_new ] = make_B2( x0 )
% Takes input value of x-coordinate of first control point in Bezier Spline
%creation.  The output is a 8x1 new version of B2.

B = [0.284 1.0 0.305 0.7 0.32 0.4 0.32 0.0;
      0.32 0.0 x0 0.3 0.34 0.7 0.32 1.0];

for k=1:2
    bx = 3*(B(k,3) - B(k,1));
    cx = 3*(B(k,5) - B(k,3)) - bx;
    dx = B(k,7) - B(k,1) - bx - cx;

    by = 3*(B(k,4) - B(k,2));
    cy = 3*(B(k,6) - B(k,4)) - by;
    dy = B(k,8) - B(k,2) - by - cy;

    %setup coeff as "cell array vector"
    xt{k} = [B(k,1), bx, cx, dx];  %each "element" in xt and xt is a 1x4
    yt{k} = [B(k,2), by, cy, dy];
end
[ xval yval ] = bezier_poly(xt,yt,2);
B2x_new = xval(2,:);
B2y_new = yval(2,:);

a = [.284 .305 .317 .32 .335 .338 .32];
b = [1 .65 .3 0 .25 .6 1];
for k = 1:2
    plot(xval(k,1:101),yval(k,1:101),'r',a,b);
    axis([.26 .38 -.1 1.1])
    hold on
end
end
```

The previous function allows us to vary the value of our x-coordinate of our first control point. Defining x0 = 0.345 for example is a much better fit for B_2. However we want to generate a curve that minimizes the distance between $B_2$and our original data points. Let's look at this error numerically rather than graphically. The following function err, considers the sum of squares difference between data points (0.335, 0.25),(0.338, 0.6) and the associated B_2spline when t = 0.3 and t= 0.6 respectively.

3)
```
function [s] = err( x0 )
%finds sum of square difference between data points and B2 curve.

[B2x_new B2y_new ] = make_B2( x0 );
a1 = B2x_new(31); % corresponds to t = .3
b1 = B2y_new(31);
a2 = B2x_new(61); % corresponds to t = .6
b2 = B2y_new(61);
                  %sum of squares formula
s = (a1 - 0.335).^2 + (b1 - 0.25).^2 + (a2 - 0.338).^2 + (b2 - 0.6).^2;
```

4) Now that an error function has been designated, as with any sort of approximation, we wish to *minimize* err using our gradient descent method seen before. Define the variable s as a function handle, $s = @(x0)\ err(x0)$, then call descent(s,0.38) as 0.38 is our initial $x_0$value used in creating B_2. The returned value is 0.34755231143569. What this means is that the least amount of distances (error) between our initial data points and the B_2 spline occurs when the x value of the first control point is ≈ 0.3476. This numerical result reflects to our graphical guess and check result of 0.345 that we noted earlier. The iterative process of gradient descent projected $x_{computed}$ in 455 steps. The graph below summarizes our findings.

5)



6) The curve to the farthest right is the associated B_2 spline when $x_0 = 0.38$, the initial x-coordinate value for the first control point for B_2. After 455 iterations of gradient descent, we see that error is minimized when $x_0 \approx 0.3476$. It is important to note that 455 cycles The tic/toc function reveals the elapsed time for executing descent (s,.38) of ≈39.3 second (which is quite high).