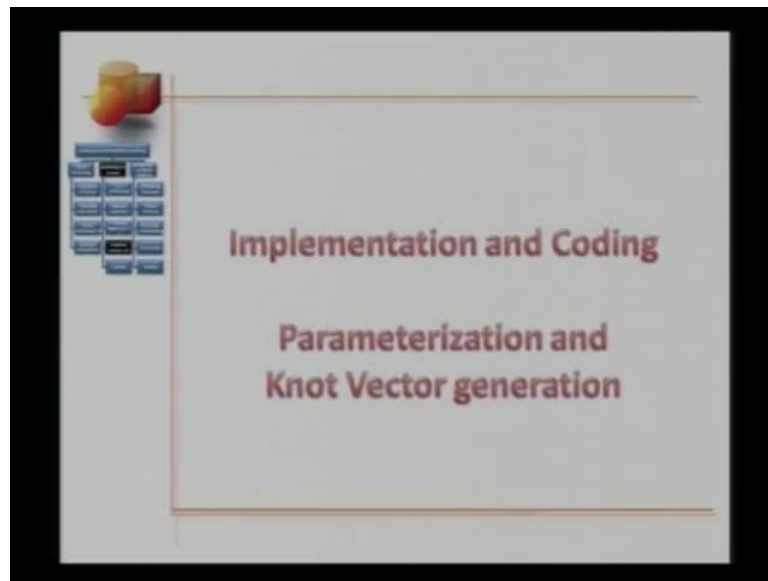


Computer Aided Engineering Design
Prof. Anupam Saxena
Department of Mechanical Engineering
Indian Institute of Technology, Kanpur

Lecture - 32

Good morning and welcome again, this is lecture number 32. Previously, we discussed a few aspects on implementation and coding for B spline curves.

(Refer Slide Time: 00:34)



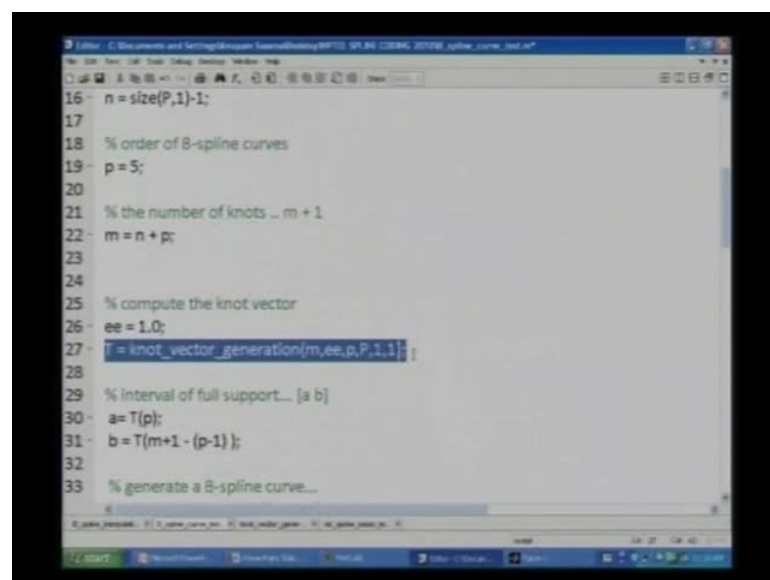
(Refer Slide Time: 00:48)

```
16 - n = size(P,1)-1;
17
18 % order of B-spline curves
19 - p = 5;
20
21 % the number of knots ... m + 1
22 - m = n + p;
23
24
25 % compute the knot vector
26 - ee = 1.0;
27 - T = knot_vector_generation(m,ee,p,P,1,1);
28
29 % interval of full support... [a b]
30 - a = T(p);
31 - b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
```

We continue with that and today we talk about parameterization and knot vector generation. Let me now continue with the matlab code. If you recall last time I had mentioned that we would want to introduce a function that would help us generate a knot vector using different parameterization schemes. What I have done is, I have written this function, and today I will run you through different pieces of the code. The function is not underscore vector or underscore generation and it requires a few parameters, I will talk about them in a while. This variable e , is the exponent that is used in various parameterizations schemes. For example, if I put e as 0, I get uniform parameterization.

If I put e as 1, I get code length parameterization and for e as half, I get centripetal parameterization. You can choose your own value for this exponent. For now, let me work with the code length parameterization. The exponent value I choose is 1 with regard to the parameters, that we would like to pass through this function. The first one pertains to the number of knots that we require. This one relates to the exponent used for different parameterizations schemes. Small p is the order of B spline basis functions and the curve capital P is a set of data points that we would need in this function, because we need to compute the code lengths of a polyline. These are two flags that relate to whether be desired to clamp the B spline curve at the first point and or at the last point.

(Refer Slide Time: 03:23)

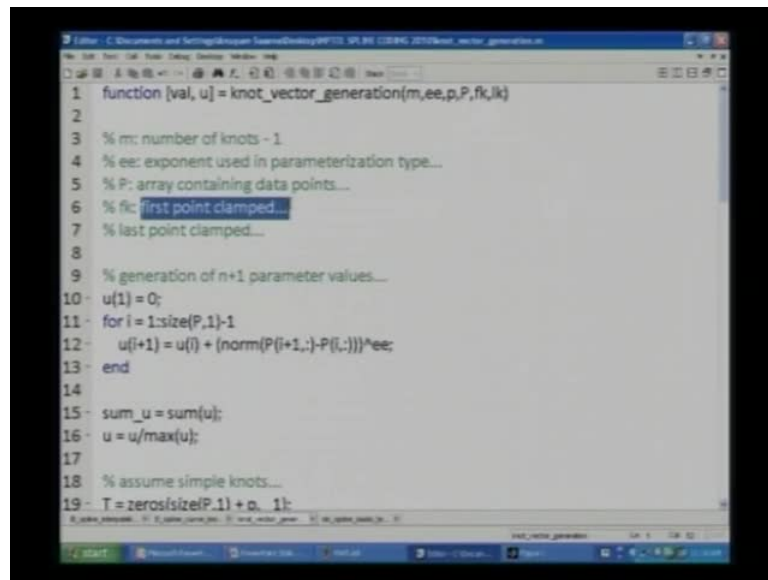
A screenshot of a MATLAB script editor window showing code for knot vector generation. The code includes comments and assignments for variables like n, p, m, ee, a, and b. The function call 'knot_vector_generation(m, ee, p, P, 1, 1)' is highlighted in blue. The code is as follows:

```
16 - n = size(P,1)-1;
17
18 % order of B-spline curves
19 - p = 5;
20
21 % the number of knots ... m + 1
22 - m = n + p;
23
24
25 % compute the knot vector
26 - ee = 1.0;
27 - y = knot_vector_generation(m, ee, p, P, 1, 1);
28
29 % interval of full support... [a b]
30 - a = T(p);
31 - b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
```

Let us now see in detail, how this function looks like? As I said m relates to the number of knots in fact m is equal to the number of knots minus 1. e is the exponent used,

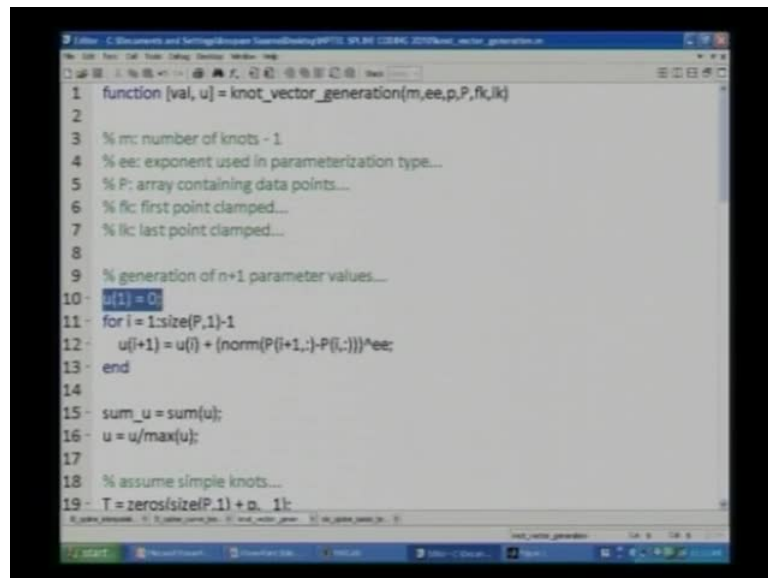
small p is the order of the B spline basis function of the curve, capital P is the array containing data points and f_k and l_k are two flags pertaining to whether we desired to clamp the B spline curve at the first point or at the last point? I should introduce l_k here, let us first work towards the generation of $n + 1$ parameter values.

(Refer Slide Time: 03:58)



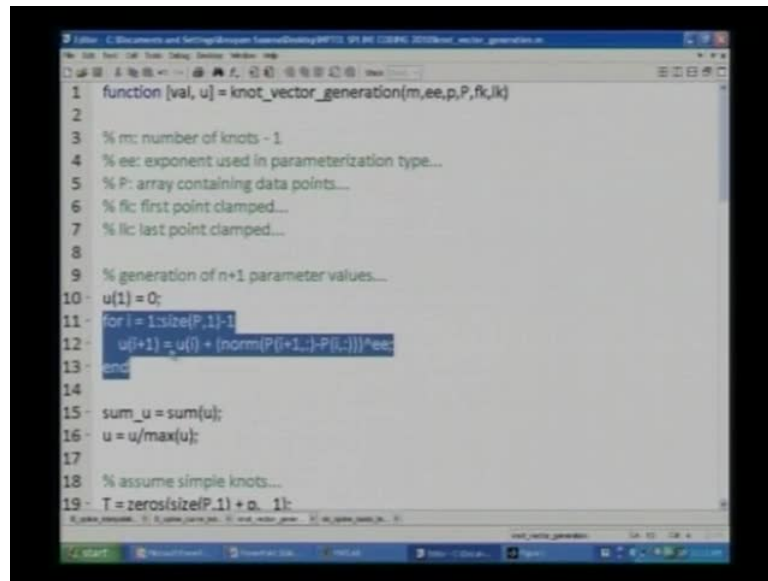
```
1 function [val, u] = knot_vector_generation(m, ee, p, P, fk, lk)
2
3 % m: number of knots - 1
4 % ee: exponent used in parameterization type...
5 % P: array containing data points...
6 % fk: first point clamped...
7 % lk: last point clamped...
8
9 % generation of n+1 parameter values...
10 u(1) = 0;
11 for i = 1:size(P,1)-1
12     u(i+1) = u(i) + (norm(P(i+1,:) - P(i,:)))^ee;
13 end
14
15 sum_u = sum(u);
16 u = u/max(u);
17
18 % assume simple knots...
19 T = zeros(size(P,1) + p, 1);
```

(Refer Slide Time: 04:15)



```
1 function [val, u] = knot_vector_generation(m, ee, p, P, fk, lk)
2
3 % m: number of knots - 1
4 % ee: exponent used in parameterization type...
5 % P: array containing data points...
6 % fk: first point clamped...
7 % lk: last point clamped...
8
9 % generation of n+1 parameter values...
10 u(1) = 0;
11 for i = 1:size(P,1)-1
12     u(i+1) = u(i) + (norm(P(i+1,:) - P(i,:)))^ee;
13 end
14
15 sum_u = sum(u);
16 u = u/max(u);
17
18 % assume simple knots...
19 T = zeros(size(P,1) + p, 1);
```

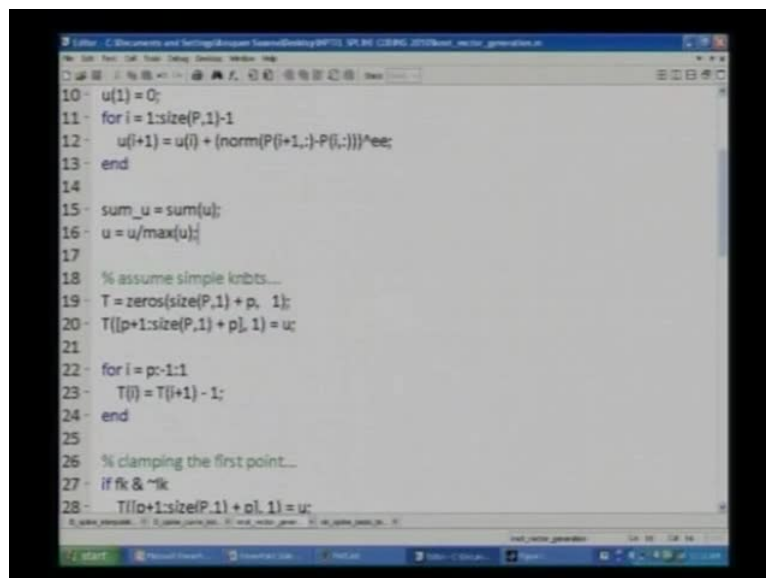
(Refer Slide Time: 04:18)



```
1 function [val, u] = knot_vector_generation(m, ee, p, P, fk, lk)
2
3 % m: number of knots - 1
4 % ee: exponent used in parameterization type...
5 % P: array containing data points...
6 % fk: first point clamped...
7 % lk: last point clamped...
8
9 % generation of n+1 parameter values...
10 u(1) = 0;
11 for i = 1:size(P,1)-1
12     u(i+1) = u(i) + (norm(P(i+1,:) - P(i,:)))^ee;
13 end
14
15 sum_u = sum(u);
16 u = u/max(u);
17
18 % assume simple knots...
19 T = zeros(size(P,1) + p, 1);
```

We said the first parameter value as 0 and we run a followed next. For i going from 1 to the number of design points minus 1, u of i plus one equals u of i plus norm of two consecutive data points i plus 1 and i data point. This length is raised to the exponent, the value of which that we have specified before. This is straight forward, what we do is we now compute the overall length of a the polyline, by adding these lengths. We normalize the parameter values as u equal u over maximum value of u .

(Refer Slide Time: 05:22)



```
10 u(1) = 0;
11 for i = 1:size(P,1)-1
12     u(i+1) = u(i) + (norm(P(i+1,:) - P(i,:)))^ee;
13 end
14
15 sum_u = sum(u);
16 u = u/max(u);
17
18 % assume simple knots...
19 T = zeros(size(P,1) + p, 1);
20 T([p+1:size(P,1) + p], 1) = u;
21
22 for i = p-1:1
23     T(i) = T(i+1) - 1;
24 end
25
26 % clamping the first point...
27 if fk & ~lk
28     T([0+1:size(P,1) + p], 1) = u;
```

(Refer Slide Time: 05:34)

```
13 - end
14
15 - sum_u = sum(u);
16 - u = u/max(u);
17
18 % assume simple knots...
19 - T = zeros(size(P,1) + p, 1);
20 - T([p+1:size(P,1) + p], 1) = u;
21
22 - for i = p-1:1
23 -     T(i) = T(i+1) - 1;
24 - end
25
26 % clamping the first point...
27 - if fk & ~lk
28 -     T([p+1:size(P,1) + p], 1) = u;
29 -     T(p) = T(p+1)-1;
30 -     for i = p-1:-1:1
31 -         T(i) = T(i+1);
```

Now, we have to consider three cases. The first case is of the simple knots. If we assume simple knots let me store the knot values in array identified by capital T. This command here, lets us initialize all the knot values to 0. Through this statement, we specified the last n plus 1 knots as the respective parameter values that we have obtained here. Now, we have the first p knots as free choices. To get them what we do is, we start subtracting negative 1 from t i plus 1, where the index i starts from p and its decremented by minus 1 every time until its value becomes 1.

(Refer Slide Time: 06:58)

```
19 - T = zeros(size(P,1) + p, 1);
20 - T([p+1:size(P,1) + p], 1) = u;
21
22 - for i = p-1:1
23 -     T(i) = T(i+1) - 1;
24 - end
25
26 % clamping the first point...
27 - if fk & ~lk
28 -     T([p+1:size(P,1) + p], 1) = u;
29 -     T(p) = T(p+1)-1;
30 -     for i = p-1:-1:1
31 -         T(i) = T(i+1);
32 -     end
33 - end
34
35 % clamping the last point...
36 - if ~fk & lk
37 -     T(1:size(P,1), 1) = u;
```

So, in affect what we doing is, we are taking the first parameter value as a reference knot. We are decrementing 1 each time to get a knot on the left. Now, this is for simple knots. Case two is the case, where we have to clamp are B spline curve at the first data point. Will be using an its condition if f_k equals 1 and l_k is not equal to 1, that is when we need to clamp the curve at the first knot. You would know what we need to do? That is right, we need to repeat the first p knots. To do so, we set the last n plus 1 knots as ourf parameter values. We insure that the first p knots get repeated, through this piece of code. For the index i going from p minus 1 decremented by 1 its final value is 1. T of i is equal to T of i plus 1, we can do something very similar. If we need to clamp are B spline curve at the last knot. Again they are going to be using the if statement, with the different argument.

(Refer Slide Time: 08:27)

```

28 - T([p+1:size(P,1) + p], 1) = u;
29 - T(p) = T(p+1)-1;
30 - for i = p-1:-1:1
31 -     T(i) = T(i+1);
32 - end
33 - end
34
35 % clamping the last point...
36 - if  $l_k \neq 1$ 
37 -     T([1:size(P,1)], 1) = u;
38 -     T(size(P,1)+1, 1) = T(size(P,1), 1) + 1;
39 -     for i = 2:p
40 -         T(size(P,1) + i) = T(size(P,1), 1) + 1;
41 -     end
42 - end
43
44 % clamping the first and last points...
45 - if  $f_k \neq 1$  &  $l_k$ 
46 -     % first  $p$  knots = 0;

```

This is the case when f_k is not equal to 1 and l_k equal to 1. If you observe this statement here here what we do is, we use the n plus 1 parameter values as the first n plus 1 knots. Observe this statement carefully, the subsequent knot is equal to the last parameter value plus 1. This is also equal to the last or the previous knots. Then through this for look here for index i going from 2 to p . The p is the order of the curve, all the subsequent knots get repeated, in fact they have the same value as s .

Let us go back a little bit and carefully look at this statement here. What we did was that, we initialized the last n plus 1 knots as are parameter values. Then a knot previous to this

array, which is T_p was computed as the next knot minus 1. Once we had the value of T of p , we simply repeated that knot through this following. We now come to the case where we need to clamp are B spline curve. Both are the first and the last data points. We use the f look for this, if the two flags have the value 1 each, then of course, we need to repeat the first p knots.

(Refer Slide Time: 10:55)

```

41- end
42- end
43
44 % clamping the first and last points...
45- if fk & 1
46- % first p knots = 0;
47- for i = 1:p
48-     T(i) = 0;
49- end
50
51- % last p knots = 1;
52- for i = 1:p
53-     T(size(P,1) + i) = 1;
54- end
55
56- % remaining size(P,1) + p - 2p knots...
57- % average p - 1 parameter values at a time
58

```

(Refer Slide Time: 11:13)

```

50
51- % last p knots = 1;
52- for i = 1:p
53-     T(size(P,1) + i) = 1;
54- end
55
56- % remaining size(P,1) + p - 2p knots...
57- % average p - 1 parameter values at a time
58
59- for i = 1:size(P,1) + p - 2*p
60-     sump = 0;
61-     for j = 1:p-1
62-         sump = sump + u(i + j);
63-     end
64-     T(p+i, 1) = sump/(p-1);
65- end
66
67- end

```

We set the values of this knots as 0, we also need to repeat the last p knots. We set the value of these knots as 1 and the number of knots that remain are the number of data

points plus p minus $2p$, which is the number of data points minus p knots, where p is the order of a B spline basis function. To compute the remaining interior knots, as we have discussed before the average $p - 1$ parameter values at a time.

(Refer Slide Time: 12:03)

```

52-   top1 = 1:p
53-   T(size(P,1) + 1) = 1;
54-   end
55-
56-   % remaining size(P,1) + 1 - 2p knots...
57-   % average p - 1 parameter values at a time
58-
59-   for i = 1:size(P,1) + p - 2*p
60-       sump = 0;
61-       for j = 1:p-1
62-           sump = sump + u(i + j);
63-       end
64-       T(p+i, 1) = sump/(p-1);
65-   end
66-
67- end
68-
69- val = T';
70-

```

(Refer Slide Time: 13:26)

```

1  function [val, u] = knot_vector_generation(m, ee, p, P, fk, lk)
2
3  % m: number of knots - 1
4  % ee: exponent used in parameterization type...
5  % P: array containing data points...
6  % fk: first point clamped...
7  % lk: last point clamped...
8
9  % generation of n+1 parameter values...
10 u(1) = 0;
11 for i = 1:size(P,1)-1
12     u(i+1) = u(i) + (norm(P(i+1,:) - P(i,:)))^ee;
13 end
14
15 sum_u = sum(u);
16 u = u/max(u);
17
18 % assume simple knots...
19 T = zeros(size(P,1) + p, 1);

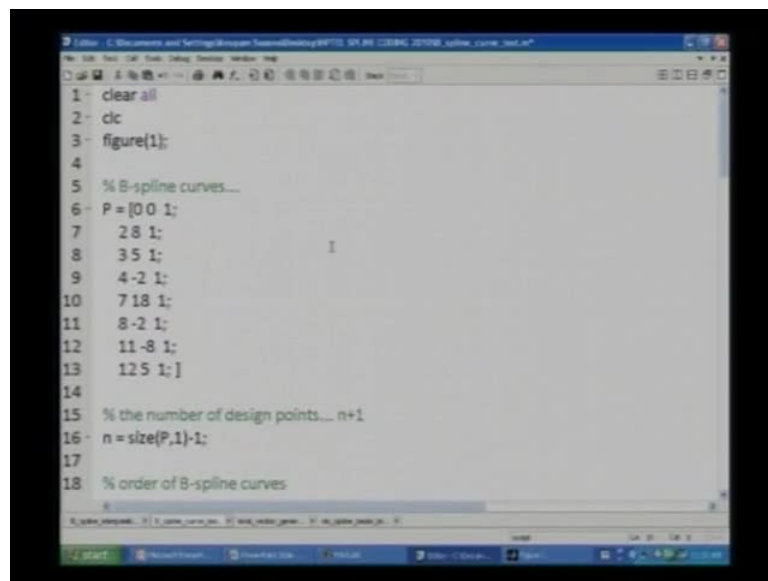
```

So, we use a nested value for this the index. i goes from 1 to the number of data points minus the order of the curve. There is a variable here sum_p , which we set to 0 for another index j going from 1 to p minus 1 sum_p is equal to sum_p plus u of i plus j index j is coming from here. Index i coming from here, with index i as i reference. We compute the

sump of the next p minus 1 parameters. Once we have that, we compute the knot value T of p plus i which is equal to this sump here over p minus 1. Finally, we returned the knot vector fact to the main function.

We may also think of returning the parameter values, which we may or may not be use in the main function. It may have been a little boring for you, you have to go through the code with me, but I will assure you once you start coding yourself, you will enjoy a great deal. Let us take a look now at few examples. Let us go back to our main function, a little comment before I show you the examples. That once we have the knot vector we can determine the interval of full support, which is given by a and b . Variable a is given by the p th knot in the knot vector and variable b is given by n plus 1 minus p minus 1 knot in the vector.

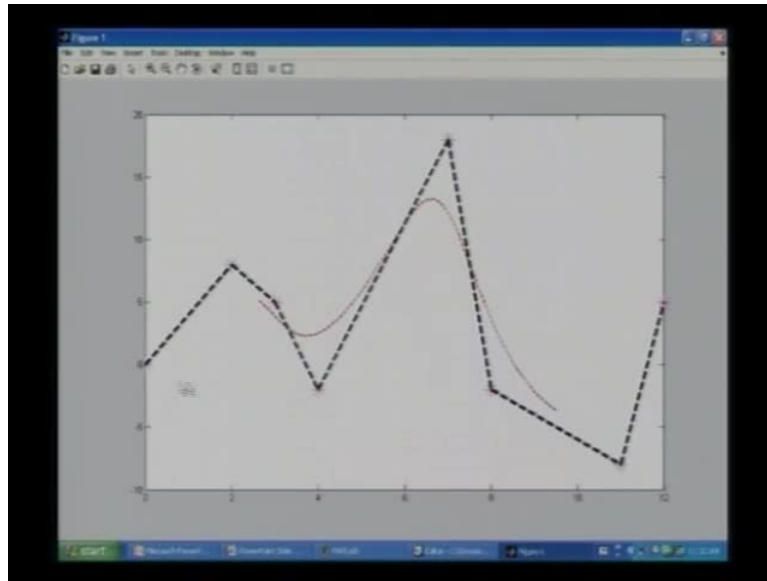
(Refer Slide Time: 14:44)



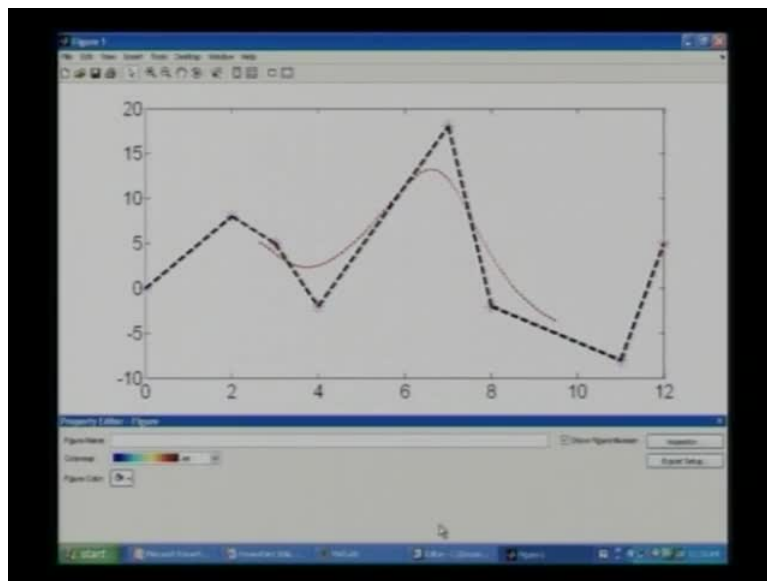
```
1 - clear all
2 - clc
3 - figure(1);
4
5 % B-spline curves...
6 P = [0 0 1;
7       2 8 1;
8       3 5 1;
9       4 -2 1;
10      7 18 1;
11      8 -2 1;
12      11 -8 1;
13      12 5 1;]
14
15 % the number of design points... n+1
16 n = size(P,1)-1;
17
18 % order of B-spline curves
```

We execute this code now, for different values of exponents. We are not interested in clamping are B spline curve at this time. So, we set the flags to 0. Let us also consider uniform parameterization in knot vector generation. Now, this is how a curve, which is free from both ends looks like. Let me change the properties of this to set the background as white and also to raise the concise of these numbers here.

(Refer Slide Time: 15:11)



(Refer Slide Time: 15:31)



The dashed black line show the control polyline and the dashed red line shows the B spline curve. Let us experiment further now. We set the exponent value as 1, which will allow us to work the code length parameterization and we execute the code again.

(Refer Slide Time: 15:52)

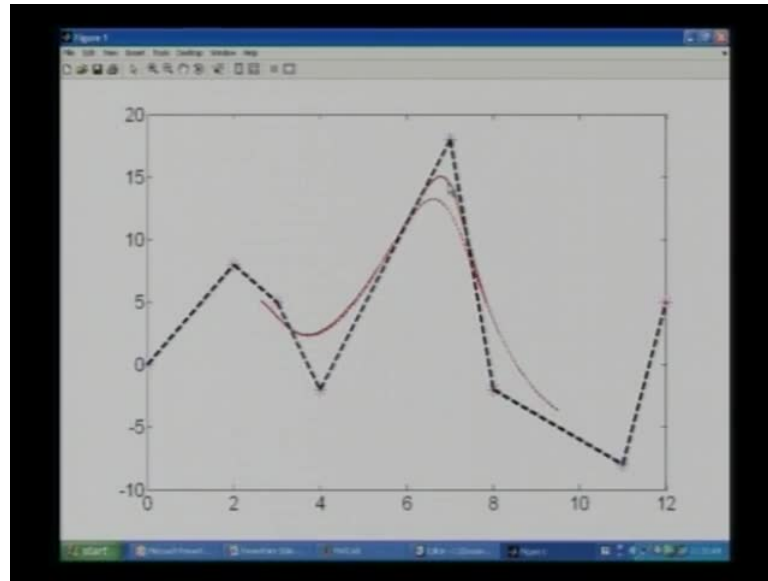
```
16 - n = size(P,1)-1;
17
18 % order of B-spline curves
19 - p = 5;
20
21 % the number of knots .. m + 1
22 - m = n + p;
23
24
25 % compute the knot vector
26 - ee = 0,0;
27 - T = knot_vector_generation(m,ee,p,P,0,0);
28
29 % interval of full support... [a b]
30 - a = T(p);
31 - b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
```

(Refer Slide Time: 16:07)

```
71
72 - end
73
74
75 % plotting the B-spline curve...
76 - plot(x,y,'r-', 'linewidth',2); hold on
77 - plot(P(:,1),P(:,2),'m*', 'MarkerSize',16);
78
79 % % plotting the Bezier curve...
80 % plot(bex,bey,'m-', 'linewidth',2); hold on
81
82 % plot the control polyline...
83 - for i = 1:n
84 -     line([P(i,1) P(i+1,1)], [P(i,2) P(i+1,2)], 'color', 'k', 'linewidth', 3, 'linestyle', '-');
85 - end
86
87
88
```

We change the properties a little bit, we are now going to be plotting this B spline curve using solid red line. You would appreciate this change. This was the initial curve and this is a new curve. The shape of the curve is expected to change, if you use different parameterization schemes. How about using the centripetal parameterization?

(Refer Slide Time: 16:22)

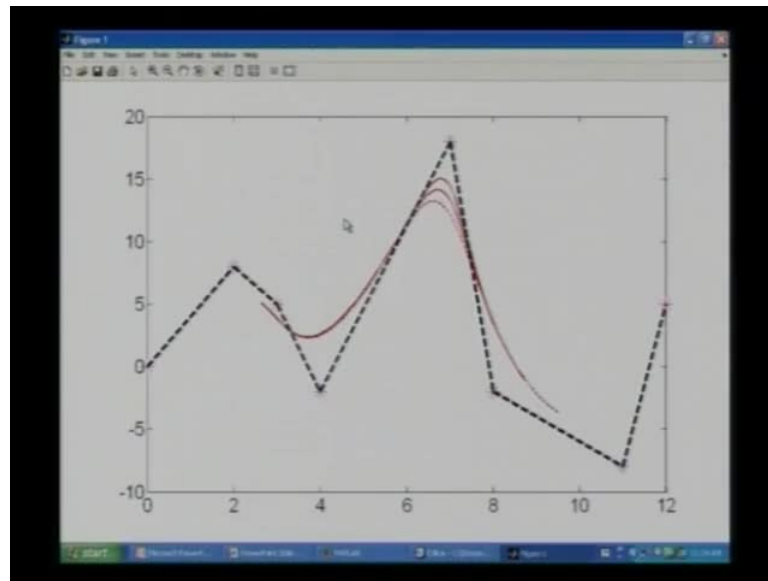


(Refer Slide Time: 16:49)

```
23
24
25 % compute the knot vector
26 ee = 1.0;
27 T = knot_vector_generation(m,ee,p,P,0,0);
28
29 % interval of full support... [a b]
30 a = T(p);
31 b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
34 % plot the corresponding Bezier curve...
35
36 for i = 1:150
37     t1 = (i-1)/149;
38     t = (1-t1)*a + t1*b - 0.000001;
39
40     if (t < a)
```

Here we set the value to 0.5. Let us retain these flag values as 0 each. Let us plot the curve using a different column. Let us say blue. You would appreciate that the curves do change in chain, is just that it is not clear to us, how the shape change will occur, if we switch between different schemes for knot vector generation? Let us maintain a single parameterization scheme.

(Refer Slide Time: 17:01)



(Refer Slide Time: 17:15)

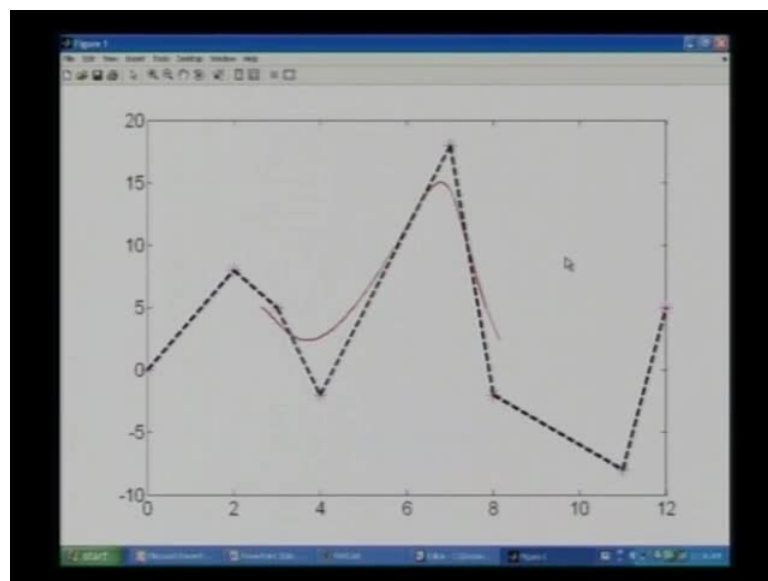
```
65- c = factorial(n)/( factorial(k-1)* factorial( n - (k-1) ) );
66- c = c*( (1-t1)^(k-1) )*(t1)^(n - (k-1) );
67-
68- bex(i) = bex(i) + c*P(k,1);
69- bey(i) = bey(i) + c*P(k,2);
70- end
71-
72- end
73-
74-
75- % plotting the B-spline curve...
76- plot(x,y,'b-','linewidth',2); hold on
77- plot(P(:,1),P(:,2),'m*', 'MarkerSize',16);
78-
79- % % plotting the Bezier curve...
80- % plot(bex,bey,'m-', 'linewidth',2); hold on
81-
82- % plot the control polyline...
```

Let us we work with code length parameterization. Now this is a curve, which is free from both ends and on this figure. Let me super post to other cases, where we now try to clamp our curve at the first point and at the last point. To clamp a curve at the first point, let us change this block to 1.

(Refer Slide Time: 17:48)

```
21 % the number of knots ... m + 1
22 m = n + p;
23
24
25 % compute the knot vector
26 ee = 1.0;
27 T = knot_vector_generation(m,ee,p,P,0,0);
28
29 % interval of full support... [a b]
30 a = T(p);
31 b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
34 % plot the corresponding Bezier curve...
35
36 for i = 1:150
37     t1 = (i-1)/149;
38     t = (1-t1)*a + t1*b - 0.000001;
```

(Refer Slide Time: 18:17)



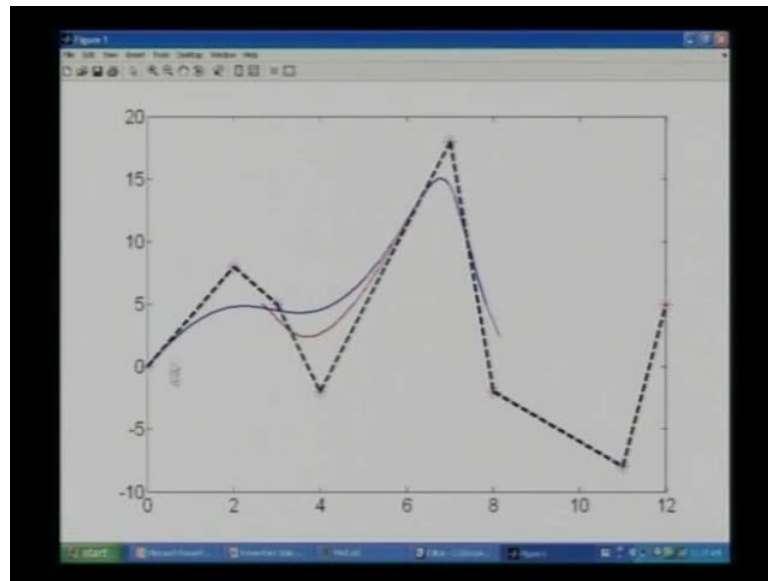
(Refer Slide Time: 18:43)

```
21 % the number of knots ... m + 1
22 m = n + p;
23
24
25 % compute the knot vector
26 ee = 1.0;
27 T = knot_vector_generation(m,ee,p,P,1,0);
28
29 % interval of full support... [a b]
30 a = T(p);
31 b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
34 % plot the corresponding Bezier curve...
35
36 for i = 1:150
37     t1 = (i-1)/149;
38     t = (1-t1)*a + t1*b - 0.000001;
```

(Refer Slide Time: 18:48)

```
66     c = c*(1-t1)^(k-1)*(t1)^(n - (k-1));
67
68     bex(i) = bex(i) + c*P(k,1);
69     bey(i) = bey(i) + c*P(k,2);
70     end
71
72 end
73
74
75 % plotting the B-spline curve...
76 plot(x,y,'b-','linewidth',2); hold on
77 plot(P(:,1),P(:,2),'m*', 'MarkerSize',16);
78
79 % % plotting the Bezier curve...
80 % plot(bex,bey,'m-', 'linewidth',2); hold on
81
82 % plot the control polyline...
83 for i = 1:n
```

(Refer Slide Time: 19:00)



Let us change the colour of the curve as well, to let us say blue and execute this code, okay? You observe what we desire.

(Refer Slide Time: 19:11)

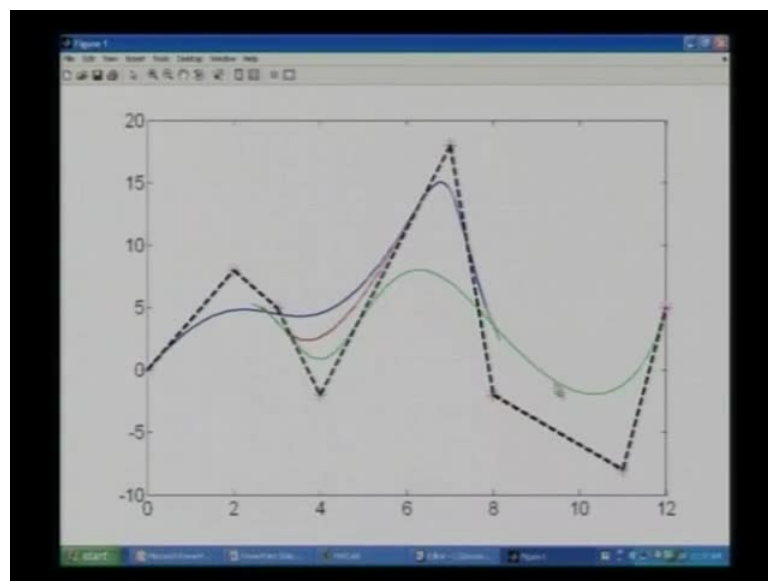
```
66- c = c*((1-t1)^(k-1))*t1^(n - (k-1));
67
68- bex(i) = bex(i) + c*P(k,1);
69- bey(i) = bey(i) + c*P(k,2);
70- end
71
72- end
73
74
75 % plotting the B-spline curve...
76- plot(x,y,'g','linewidth',2); hold on
77- plot(P(:,1),P(:,2),'m','MarkerSize',16);
78
79 % % plotting the Bezier curve...
80 % plot(bex,bey,'m-','linewidth',2); hold on
81
82 % plot the control polyline...
83- for i = 1:n
```

Let me now change this colour to green. Go back change these flags to 0 and 1 respectively. What I want now is to how my B spline curve pass through the last point?

(Refer Slide Time: 19:17)

```
23
24
25 % compute the knot vector
26 ee = 1.0;
27 T = knot_vector_generation(m,ee,p,P,0,0);
28
29 % interval of full support... [a b]
30 a = T(p);
31 b = T(m+1 - (p-1));
32
33 % generate a B-spline curve...
34 % plot the corresponding Bezier curve...
35
36 for i = 1:150
37     t1 = (i-1)/149;
38     t = (1-t1)*a + t1*b - 0.000001;
39
40     if (t < a)
```

(Refer Slide Time: 19:30)



This is how the curve looks like. It is clamped at this last data point. Finally this clamp occurs at both the end points, which is the different colour to colour this curve. This one is the final curve like here that passes through the first as well as the last point. Let us now continue with the lecture number 32 and discuss a few more very important aspects of B-spline segments and curves. The first one relates to the interpolation and the second one pertains to NURBS, non uniform rational B-splines. Let us see how we can interpolate different data points with B-spline curves?

(Refer Slide Time: 20:49)

Interpolation with B-spline curves

Given $n+1$ data points p_0, p_1, \dots, p_n fit them with a B-spline curve of given order $p \leq n$ n + 1 conditions

a set of parameters u_0, u_1, \dots, u_n may be generated
the number of knots $m + 1$ may be computed
knot vector $[t_0, t_1, \dots, t_m]$ may then be computed Basis functions known

Given $n + 1$ data points p_0, p_1, \dots, p_n , we required to design a B spline curve, that fits these data points. Let us say we choose the order of a B spline curve p , which is smaller than or equal to in number of data points minus 1. So, N here is equal to the number of data points minus 1. We can generate a set of parameter values u_0, u_1, \dots, u_n , through any scheme that we had discussed previously. Accordingly the number of knots can be computed. The length of the knot vector is $n + 1$, which is given by the order plus the number of data points $n + 1$, you know that already.

Clearly, we will have a knot vector T_0, T_1, \dots, T_N in our hands to work with with that the basis functions will be known to us. What is next? If you look at this problem, it is an inverse problem. Why do I say that? Simply because we are not directly specifying the control polyline, to design our B spline curve, rather we are specifying a set of points through our B spline curve would pass. For that we need to compute or control polyline, which is why its inverse problem. Let us investigate this further.

(Refer Slide Time: 23:17)

Interpolation with B-spline curves

REQUIRED TO FIND THE INTERPOLATING B-SPLINE CURVE

$$\mathbf{b}(t) = \sum_{i=0}^n N_{p,p+i}(t)\mathbf{b}_i$$

Control points \mathbf{b}_i are $(n+1)$ unknowns

Consider $\mathbf{p}_k = \mathbf{b}(u_k) = \sum_{i=0}^n N_{p,p+i}(u_k)\mathbf{b}_i \quad k = 0, \dots, n$

$$\mathbf{P} = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ \mathbf{p}_n \end{pmatrix} = \begin{pmatrix} N_{p,p}(u_0) & N_{p,p+1}(u_0) & N_{p,p+2}(u_0) & \dots & N_{p,n}(u_0) \\ N_{p,p}(u_1) & N_{p,p+1}(u_1) & N_{p,p+2}(u_1) & \dots & N_{p,n}(u_1) \\ N_{p,p}(u_2) & N_{p,p+1}(u_2) & N_{p,p+2}(u_2) & \dots & N_{p,n}(u_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ N_{p,p}(u_n) & N_{p,p+1}(u_n) & N_{p,p+2}(u_n) & \dots & N_{p,n}(u_n) \end{pmatrix} \begin{pmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_n \end{pmatrix} = \mathbf{N}\mathbf{B}$$

We are required to find the interpolating B-spline curve. As you would know, this curve is given by summation index i going from 0 to n of $N_{p,p+i}(t)$ times \mathbf{b}_i . p is the order of the curve and i is the last knot for which $N_{p,p+i}$ stands. \mathbf{b}_i is the usually the design point that a user would specify. In this case \mathbf{b}_i is an unknown. So, we have $n+1$ \mathbf{b}_i 's, which are unknowns. I need to find them. Now, let us consider that we know the information at different points on the B-spline curve. In other words, we know the points through which we need to interpolate to B-spline curve.

These points are given by \mathbf{p}_k . These points are known at different parameter values u_k . You plug in this parameter value here to get summation i going from 0 to n of $N_{p,p+i}(u_k)$ times \mathbf{b}_i . We know that we have or rather we are working with $n+1$ unknowns. We would need to generate $n+1$ conditions. These conditions will relate to different k values going from 0 to n . We can collate these conditions in the matrix form.

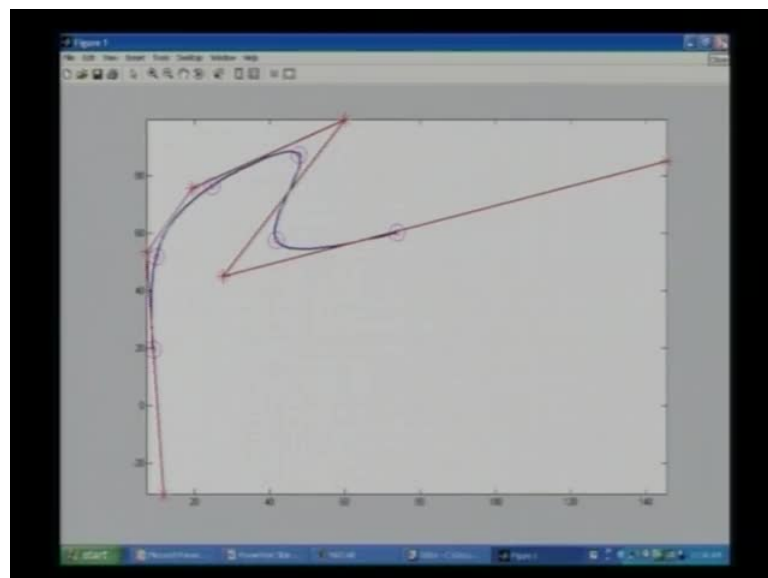
Let us say capital \mathbf{P} is equal to $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ up to \mathbf{p}_n , these are the data points available to us. That is equal to $(n+1) \times (n+1)$ matrix. The first of that is $N_{p,p}(u_0), N_{p,p+1}(u_0), N_{p,p+2}(u_0), \dots, N_{p,n}(u_0)$. The last entry in this row is $N_{p,n}(u_0)$. The second row is the same as a first row, except that we are using u_1 as a set of u_0 . The similar is the case for the third row. The basis functions are the same, but evaluated at u_2 . Likewise for the last row here, we will have all these basis functions evaluating at u_n .

Here we will have a vector corresponding to these design points b_{i0} b_{11} up to b_N . We will have to be extra careful in choosing the parameter values here. We need to ensure that to determine these design points inversion of this $n+1$ by $N+1$ matrix does not cause any problem. In other words we want this $N+1$ by $N+1$ matrix to be non singular. The right hand side in short is represented by this matrix N , which is this one times the matrix b which is this. Let us take a look at a few examples.

(Refer Slide Time: 28:21)

```
7
8 % Specifying design points...
9 while_switch = 1;
10 k = 0;
11 while while_switch
12     k = k+1;
13     [xx,yy,button] = ginput(1);
14     button =
15     P(k,1) = xx;
16     P(k,2) = yy;
17
18     plot(P(k,1),P(k,2),'mo','MarkerSize',18); hold on
19     axis([0 100 0 100]);
20
21     if button==3
22         while_switch = 0;
23     end
24 end
```

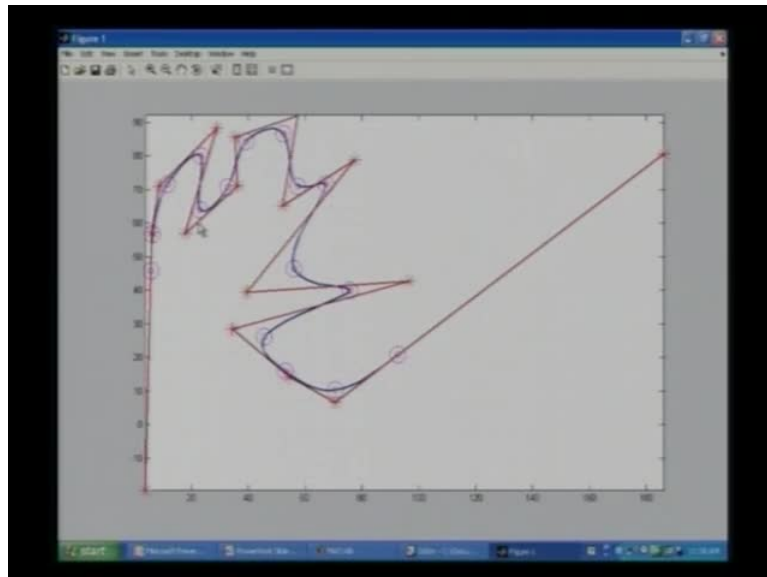
(Refer Slide Time: 28:48)



What I have done here is I have introduced a new piece of code, to interactively specified the data points. Let us not get into that and straight away work on the examples. We are working with order four B spline curves.

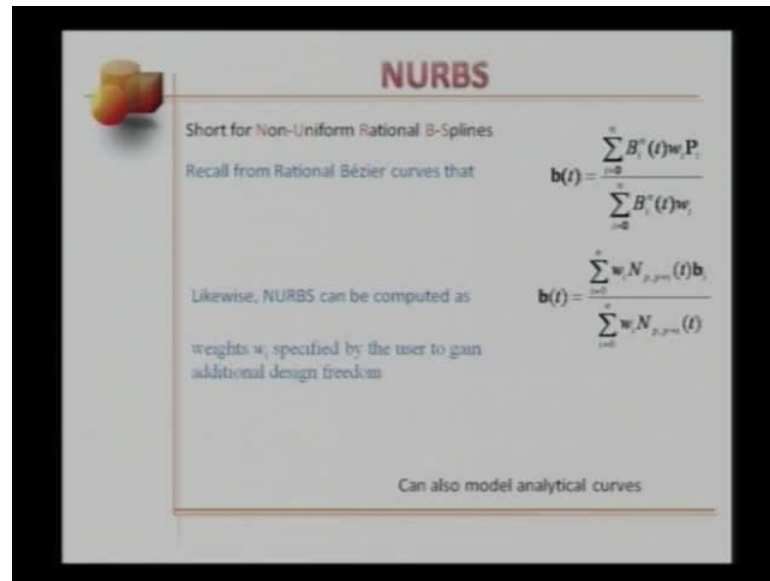
Let me magnify this figure and say specify six data points through which, I would require my B spline curve to pass. Now, the curve in blue is our B spline curve, it is a curve in full support. The polyline in red is what is determined the points in stars are the control points, that we have just determined for this B spline curve. Let us consider a few more examples. The curve in blue passes through the points that we just specified. The stars are the control points that are determined and the polyline in red joins, this control points. Let us see it can interpolate of B spline curve through a large set of data points.

(Refer Slide Time: 30:37)



I am arbitrary specifying these points here. I have specified 4 7 10 12 15 and this is a last point 16. It looks like I can get my B spline curve to pass through a set of data points, any number that I specify. Accordingly, my control points get recomputed will be time. The control polyline changes, it become as well We now get to a very important concept in B spline curves, NURBS.

(Refer Slide Time: 31:40)



It is an abbreviation for non uniform rational B splines. Recall that we had discussed rational Bezier curves quite some time ago. A rational bezier curve is given by $\mathbf{b}(t)$ equals something in the numerator and something in the denominator. A numerator is summation the index i going from 0 to N . Bernstein polynomial with index i and degree N , which is a function of parameter T times the weight w_i , which is assigned by the user corresponding to the design point P_i in the denominator. We let go of P_i in this summation.

So, we have summation, i am going from 0 to N , the N th degree Bernstein polynomial with index i function of T times the weight w_i . We had also seen that we could model different quadric functions precisely using rational Bezier models. Just that we did not have much local control to be often by Bernstein basis functions. Splines can be generalized along very similar lines. All we have to do is replace Bernstein polynomials with the corresponding B spline basis functions of order p . The design information points and from the respective weights remain the same.

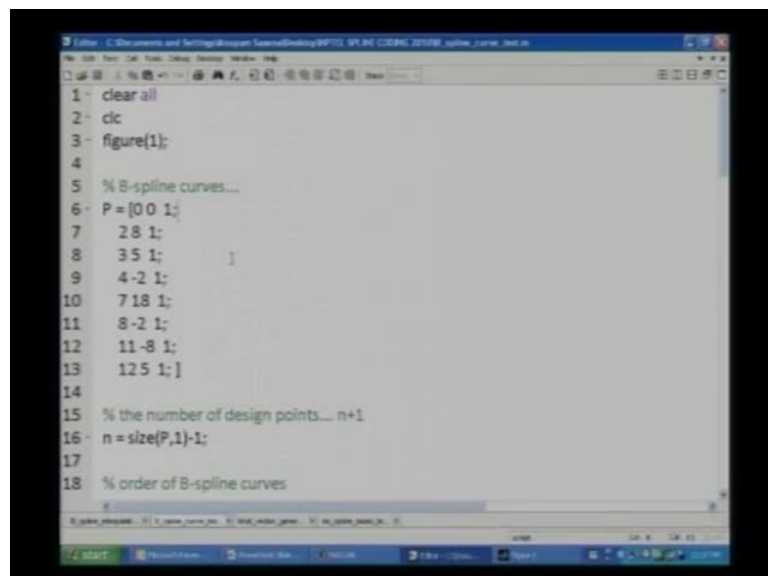
If you look at these two expressions carefully, this one here offers a great deal of freedom and choice. Both freedom in terms of additionally specifying the order of B spline curve and in terms of local control offered by the B spline basis functions $n = p + 1$. As I mentioned before the weights w_i , I can help a user to gain additional design

freedom. Let us try to understand each term in this expression non uniform rational B splines non uniform, because we can have the knot vector to be non uniform.

Implying that the knots may not be uniformly placed. In another words knots spans may not be of equal lengths by rational because of this expression you work in the fractions. That is why rational. You would have guessed why B splines because we are working with B spline basis functions. Although this is a field in itself I will only highlight some salient features of NURBS in this lecture. If I sit the weight corresponding to the i control point equal to 0, it is natural for us to expect that the location of b_i the corresponding design point will not affect the shape of the curve.

For larger values of a weight w_i b_i would expect just in the case with rational Bezier curves, that a B spline curve will get pushed towards the corresponding design point b_i . This model of course, offers great flexibility in design because we use B spline basis functions. NURBS posses local shape control and in here all properties of B spline basis functions and therefore, B spline curves. We are widely used in tree form curve design and can also be used the model analytical curves. Let us consider some examples, I have modified the code that I have been developing.

(Refer Slide Time: 38:06)

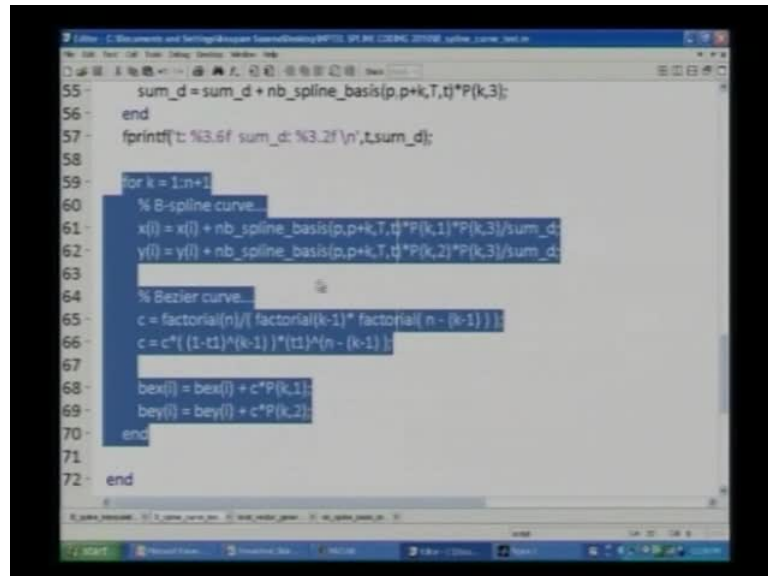


```
1 - clear all
2 - tic
3 - figure(1);
4
5 % B-spline curves...
6 - P = [0 0 1;
7       2 8 1;
8       3 5 1;
9       4 -2 1;
10      7 18 1;
11      8 -2 1;
12      11 -8 1;
13      12 5 1;]
14
15 % the number of design points... n+1
16 - n = size(P,1)-1;
17
18 % order of B-spline curves
```

A little bit this array p , that had the x co ordinates in the first column and the y co ordinates in the second column. Now, has the weights associated in the third column. At

this time all the weights are initialized to 1. As you would expect with all weights has 1 we will be getting our B spline curve.

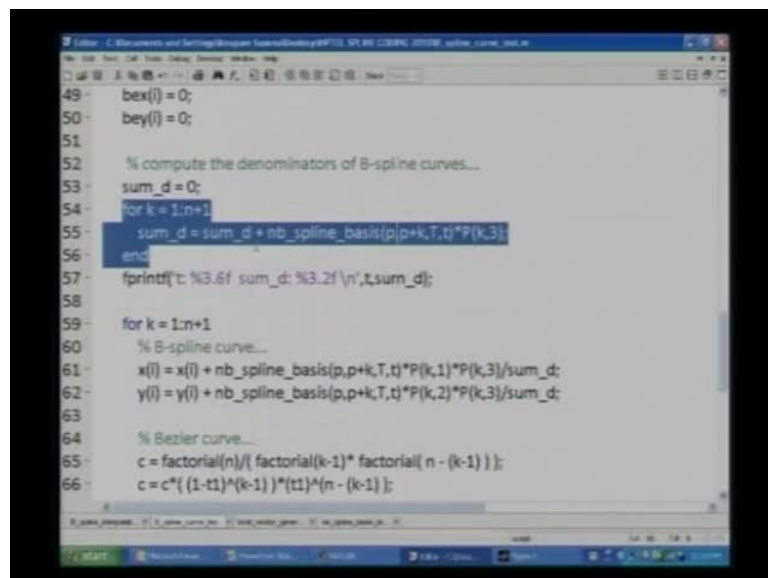
(Refer Slide Time: 38:59)



```
55- sum_d = sum_d + nb_spline_basis(p,p+k,T,t)*P(k,3);
56- end
57- fprintf('t: %3.6f sum_d: %3.2f \n',t,sum_d);
58-
59- for k = 1:n+1
60-     % B-spline curve...
61-     x(i) = x(i) + nb_spline_basis(p,p+k,T,t)*P(k,1)/sum_d;
62-     y(i) = y(i) + nb_spline_basis(p,p+k,T,t)*P(k,2)/sum_d;
63-
64-     % Bezier curve...
65-     c = factorial(n)/(factorial(k-1)*factorial(n-(k-1)));
66-     c = c*(1-t)^(k-1)*t^(n-(k-1));
67-
68-     bex(i) = bex(i) + c*P(k,1);
69-     bey(i) = bey(i) + c*P(k,2);
70- end
71-
72- end
```

Over here I have multiplied the definition of a B spline curve to include the weight information. So, here I multiply to respective design points with the weights and I compute the denominator sum underscore d before.

(Refer Slide Time: 39:29)

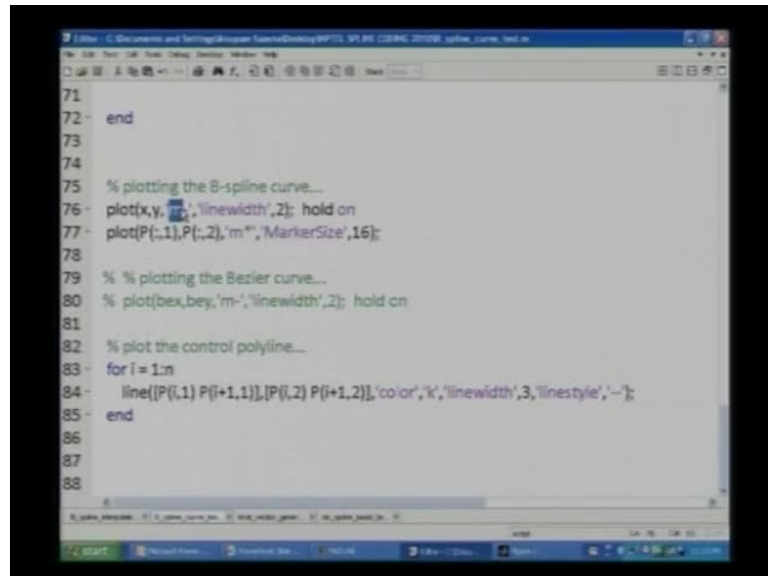


```
49- bex(i) = 0;
50- bey(i) = 0;
51-
52- % compute the denominators of B-spline curves...
53- sum_d = 0;
54- for k = 1:n+1
55-     sum_d = sum_d + nb_spline_basis(p,p+k,T,t)*P(k,3);
56- end
57- fprintf('t: %3.6f sum_d: %3.2f \n',t,sum_d);
58-
59- for k = 1:n+1
60-     % B-spline curve...
61-     x(i) = x(i) + nb_spline_basis(p,p+k,T,t)*P(k,1)/sum_d;
62-     y(i) = y(i) + nb_spline_basis(p,p+k,T,t)*P(k,2)/sum_d;
63-
64-     % Bezier curve...
65-     c = factorial(n)/(factorial(k-1)*factorial(n-(k-1)));
66-     c = c*(1-t)^(k-1)*t^(n-(k-1));
```

This is how I compute the denominator? Sum under code d equals sum under code d plus B spline basis function of order p, the last knot as p plus k. the knot vector T, that we

now know how to generate using different parameterization schemes and the parameter value. p_k comma 3 corresponds the weight of the k design point.

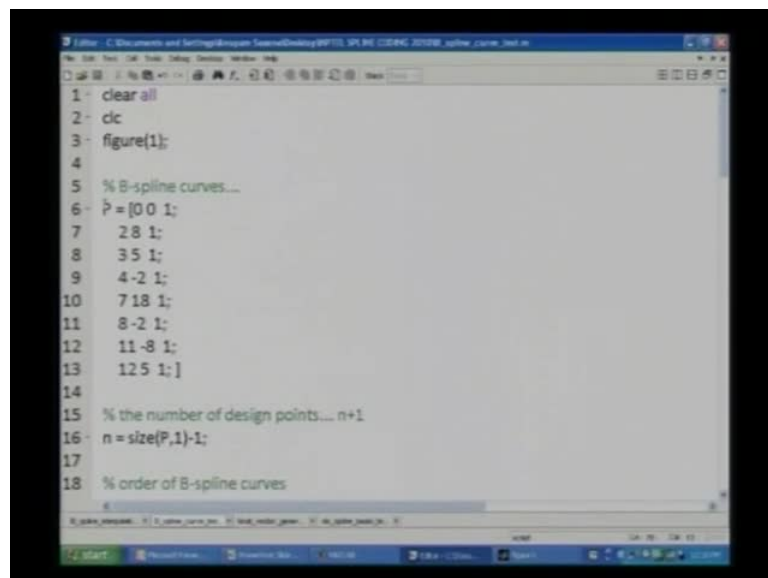
(Refer Slide Time: 40:06)



```
71 end
72 end
73
74
75 % plotting the B-spline curve...
76 plot(x,y,'r','linewidth',2); hold on
77 plot(P(:,1),P(:,2),'m*','MarkerSize',16);
78
79 % % plotting the Bezier curve...
80 % plot(bex,bey,'m-','linewidth',2); hold on
81
82 % plot the control polyline...
83 for i = 1:n
84     line([P(i,1) P(i+1,1)], [P(i,2) P(i+1,2)], 'color','k','linewidth',3,'linestyle','-');
85 end
86
87
88
```

Let us plot the first curve with the red solid line.

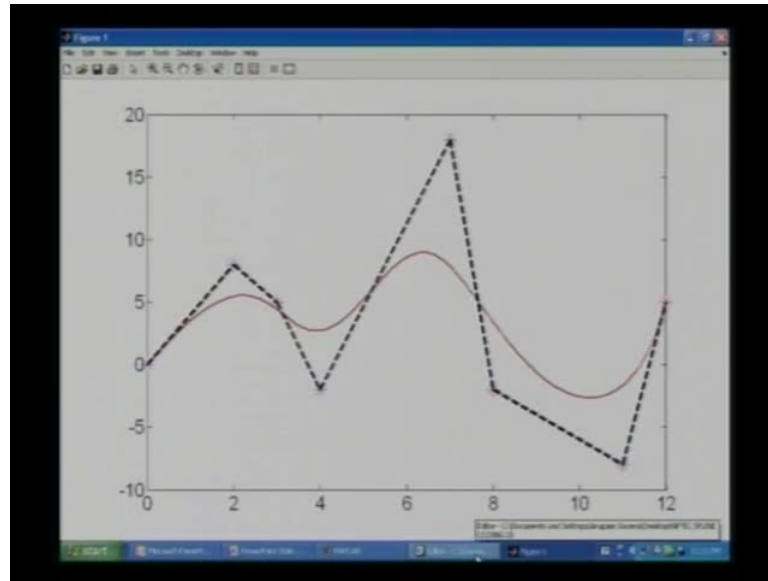
(Refer Slide Time: 40:14)



```
1 clear all
2 clc
3 figure(1);
4
5 % B-spline curves...
6 P = [0 0 1;
7     2 8 1;
8     3 5 1;
9     4 -2 1;
10    7 18 1;
11    8 -2 1;
12    11 -8 1;
13    12 5 1;]
14
15 % the number of design points... n+1
16 n = size(P,1)-1;
17
18 % order of B-spline curves
```

These are the design points we have, you are working with code length parameterization and we would like to clamp our B spline curve at both the first and the last control points. Let us see how the curves looks like?

(Refer Slide Time: 40:38)



Let me change the properties of this figure, so that it becomes much clearer. This is our original vector. Let us experiment with this point here as a reference. First, second, third, fourth, fifth point here.

(Refer Slide Time: 41:21)

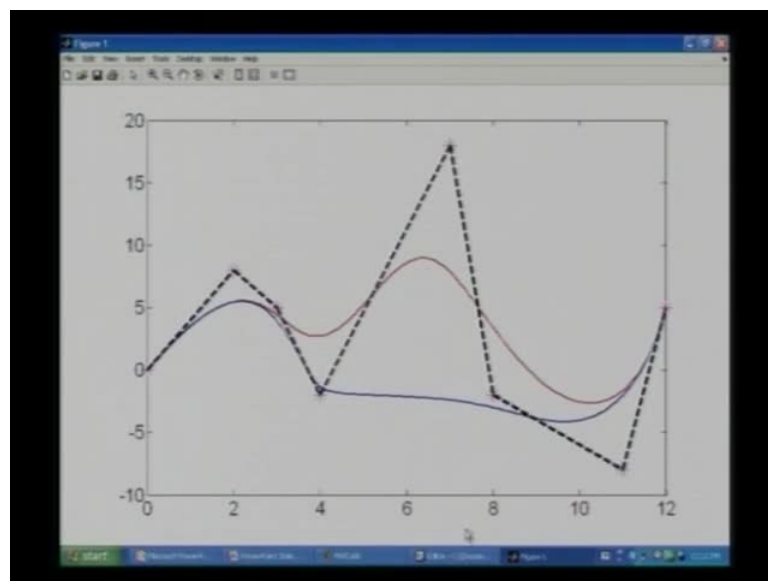
```
1 - clear all
2 - clc
3 - figure(1);
4
5 % B-spline curves...
6 P = [0 0 1;
7       2 8 1;
8       3 5 1;
9       4 -2 1;
10      7 18 0;
11      8 -2 1;
12      11 -8 1;
13      12 5 1;]
14
15 % the number of design points... n+1
16 n = size(P,1)-1;
17
18 % order of B-spline curves
```

Let us go back the code and change the weight corresponding to this point; first, second, third, fourth and fifth. Let us first set the weight to 0 and see what happens?

(Refer Slide Time: 41:37)

```
71
72 end
73
74
75 % plotting the B-spline curve...
76 plot(x,y,'-',linewidth,2); hold on
77 plot(P(:,1),P(:,2),'m^+',MarkerSize,16);
78
79 % plotting the Bezier curve...
80 % plot[bex,bey,'m-',linewidth,2]; hold on
81
82 % plot the control polyline...
83 for i = 1:n
84     line([P(i,1) P(i+1,1)], [P(i,2) P(i+1,2)], 'color','k','linewidth',3,'linestyle','-');
85 end
86
87
88
```

(Refer Slide Time: 41:40)



Let us use a different colour to plot this colour. It looks like the new curve here in blue is not affected by the position of this point at all. For w equal 0, this is a new curve for the corresponding weight equals 1, the curve gets slightly pushed towards this point. Let us try to see this happens.

(Refer Slide Time: 42:22)

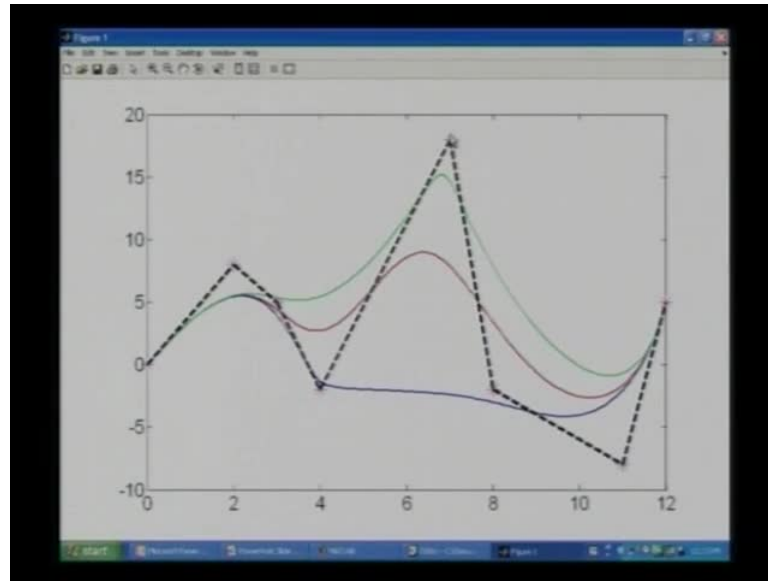
```
71
72 end
73
74
75 % plotting the B-spline curve...
76 plot(x,y,'g','linewidth',2); hold on
77 plot(P(:,1),P(:,2),'m','MarkerSize',16);
78
79 % plotting the Bezier curve...
80 % plot[bex,bey,'m','linewidth',2]; hold on
81
82 % plot the control polyline...
83 for i = 1:n
84     line([P(i,1) P(i+1,1)], [P(i,2) P(i+1,2)], 'color','k', 'linewidth',3, 'linestyle','-');
85 end
86
87
88
```

(Refer Slide Time: 42:30)

```
1 clear all
2 clc
3 figure(1);
4
5 % B-spline curves...
6 P = [0 0 1;
7     2 8 1;
8     3 5 1;
9     4 -2 1;
10    7 18 5;
11    8 -2 1;
12    11 -8 1;
13    12 5 1;]
14
15 % the number of design points... n+1
16 n = size(P,1)-1;
17
18 % order of B-spline curves
```

You plot the next curve using the green colour. And we change the weight of this point to let say 5. Indeed if I increase the weight of this point the curve gets locally attracted towards it.

(Refer Slide Time: 42:39)



(Refer Slide Time: 42:51)

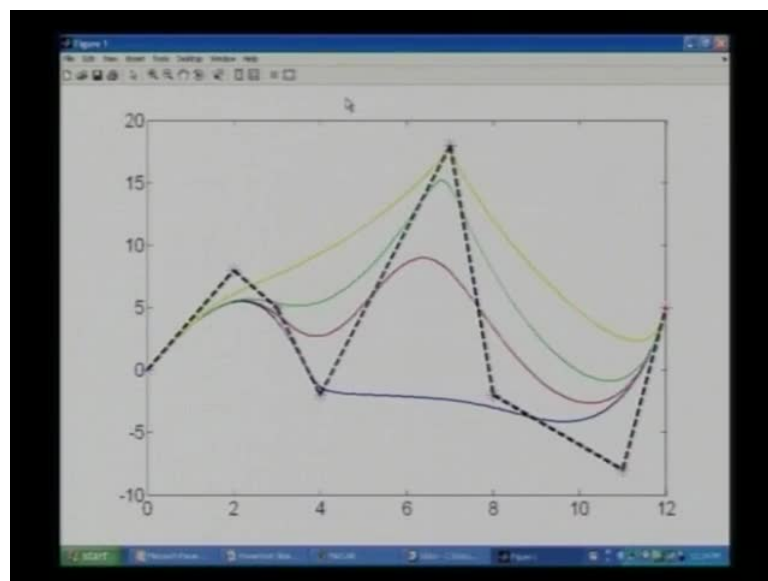
```
1 - clear all
2 - clc
3 - figure(1);
4
5 % B-spline curves...
6 - P = [0 0 1;
7       2 8 1;
8       3 5 1;
9       4 -2 1;
10      7 18 100;
11      8 -2 1;
12      11 -8 1;
13      12 5 1;]
14
15 % the number of design points... n+1
16 - n = size(P,1)-1;
17
18 % order of B-spline curves
```

Let us arbitrary make the weight as 100. So, very high weight, but what is the harm in the experimenting? Let us use the yellow colour with thicker line width to plot this new curve. It seems that the resulting curve almost passes through this point. Let us magnify this region here.

(Refer Slide Time: 43:04)

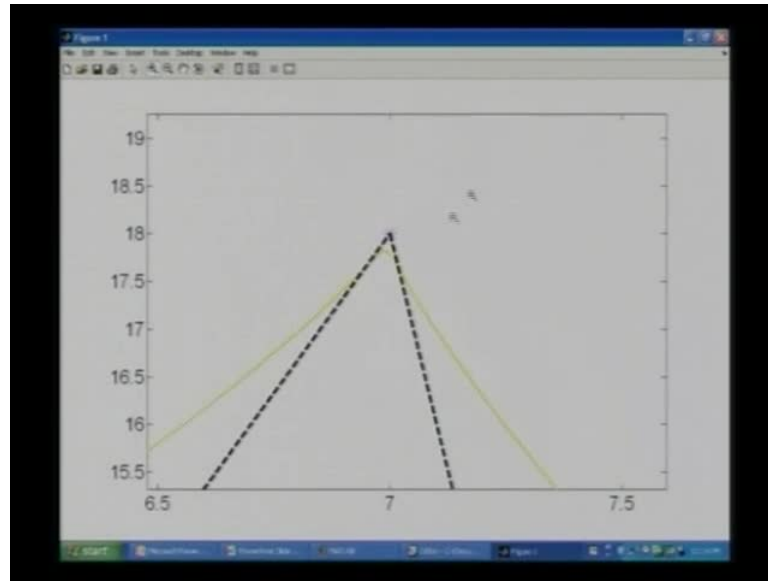
```
71
72 end
73
74
75 % plotting the B-spline curve...
76 plot(x,y,'linewidth',3); hold on
77 plot(P(:,1),P(:,2),'m*', 'MarkerSize',16);
78
79 % plotting the Bezier curve...
80 % plot(bex,bey,'m-', 'linewidth',2); hold on
81
82 % plot the control polyline...
83 for i = 1:n
84     line([P(i,1) P(i+1,1)], [P(i,2) P(i+1,2)], 'color','k', 'linewidth',3, 'linestyle','-');
85 end
86
87
88
```

(Refer Slide Time: 43:15)

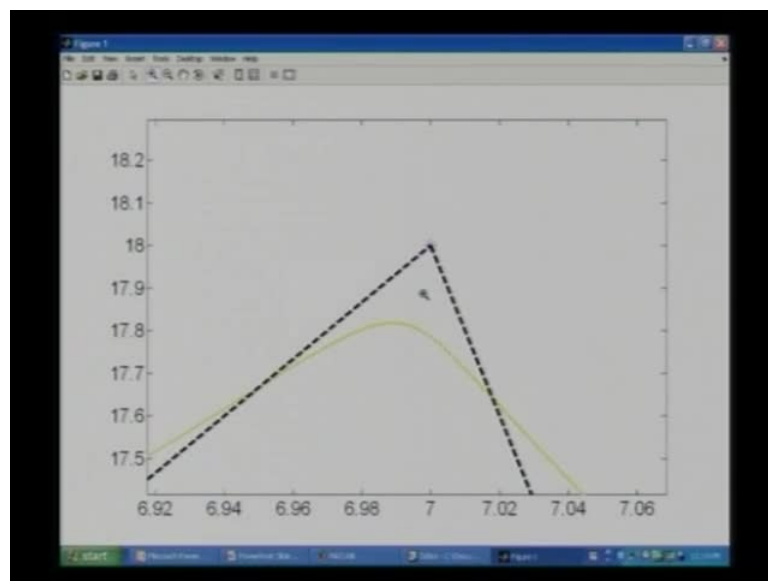


As I said, it almost passes through this point. It does not actually passes this point. What have shown now is, how to change the shape of the curve corresponding to the weight of the single point? Once you are ready with your own code, we can experiment how it can play with the shape of these B spline curves? Now NURBS by changing the positions and the weights corresponding to each point. There is still alot that if you want we can learn in this serial. For now let us stop our discussion on NURBS here. Before I leave you, I like to mention a few words about B spline basis functions and Bernstein polynomials.

(Refer Slide Time: 43:34)

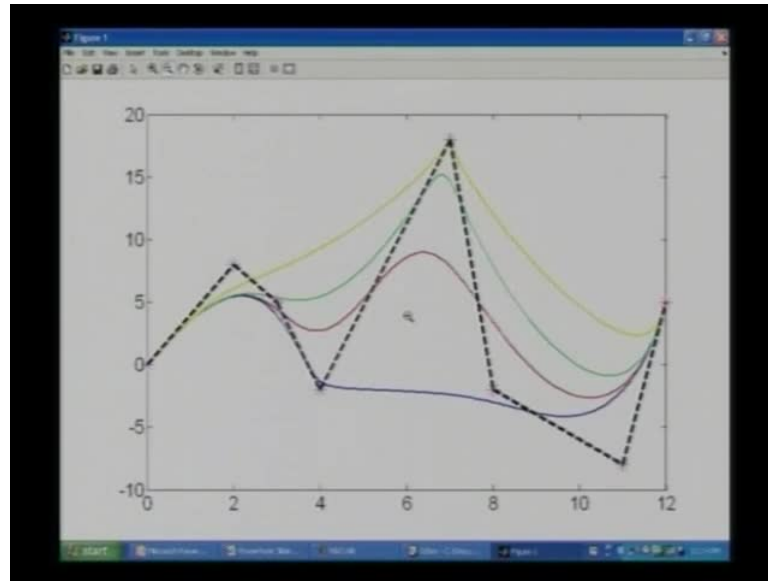


(Refer Slide Time: 43:36)



It looks like you can convert B spline basis functions to Bernstein polynomials. For specific cases for an order p curve or B spline basis function, if you repeat the first knot with value 0 p times and if you do the same. That is if you repeat the last knot with the value 1 p times and if you consider p B spline basis functions, which is the same as the control points is... so happen, that the number of knots given by m plus 1 will be such that m will be given by n plus p n is p from here. So, it is p plus p which is $2p$ and in this case the B spline basis functions will degenerate to Bernstein polynomials. Let us take a quick look at one of the examples.

(Refer Slide Time: 43:58)



(Refer Slide Time: 45:01)

B-spline and Bernstein polynomials

For an order p curve ...

- Repeat the first knot '0' p times
- Repeat the last knot '1' p times
- Consider $n + 1 = p$ B-spline basis functions/ Control points
- \Rightarrow number of knots $(m + 1)$; $m = n + p = p + p = 2p$

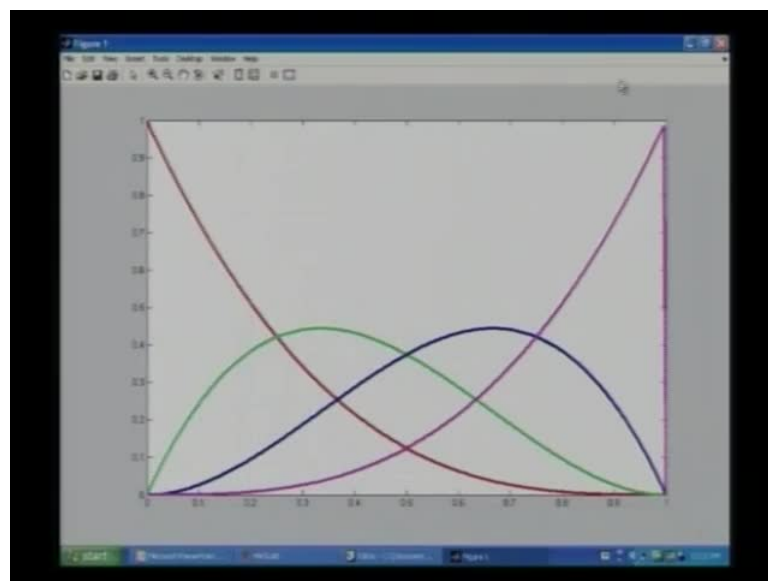
THE B-SPLINE BASIS FUNCTIONS DEGENERATE TO BERNSTEIN POLYNOMIALS

Let us see how order four B spline basis functions degenerate to degree three Bernstein polynomials. So, for that we would need four B spline basis functions and the total number of knots will be 4 plus 4. Of these let say the first four knots at the value 0 and the second set of four knots as the value 1. Let us plot the corresponding B spline basis functions.

(Refer Slide Time: 46:39)

```
1
2- clc, figure(1), cla
3- clear all
4
5 % order: m
6- m = 4;
7
8 %T: KNOT VECTOR
9 %T = [0 1 2 3 4 5 6 7 8 9 10]/10;
10- T = [0 0 0 0 10 10 10 10]/10;
11
12 % plotting nb_spline_basis functions
13
14- for i = 1:200
15-     t = (i-1)/199;
16
17-     x(i) = [1-t]*T(1) + t*T(end);
18
19 % BSBF: B Spline basis functions
```

(Refer Slide Time: 47:26)



As you would note, they are precisely the respective Bernstein polynomials $b_{0,3}$, $b_{1,3}$, $b_{2,3}$ and $b_{3,3}$. You might want to work it out mathematically as to how this happens?