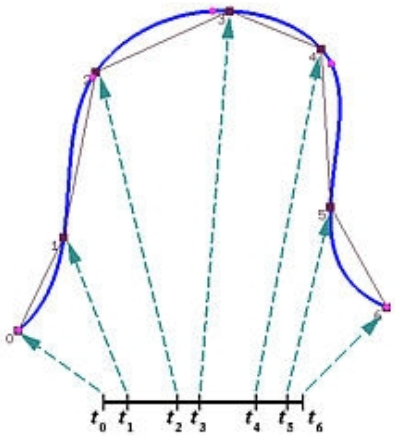
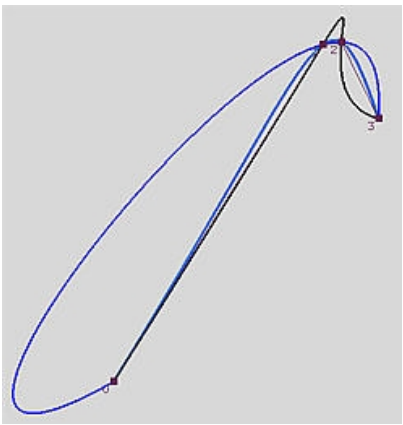
B-Spline Interpolation and Approximation

Parameter Selection Overview

The input to a B-spline interpolation/approximation algorithm usually consists of a set of data points. Thus, the first step is to find a set of parameters that can "fixed" these points at certain values. More precisely, if the data points are $\mathbf{D}_0$, ..., $\mathbf{D}_n$, then n+1 parameters t_0, ..., t_n in the domain of the curve must be found so that data point $\mathbf{D}_k$ corresponds to parameter t_k for k between 0 and n. This means that if $\mathbf{C}(u)$ is a curve that passes through all data points in the given order, then we have $\mathbf{D}_k = \mathbf{C}(t_k)$ for all $0 \le k \le n$. In the figure below, we have seven data points (*i.e.*, $n = 6$) and seven parameters must be found to setup the desired correspondence.



There are infinite number of possibilities for selecting these parameters. For example, we can evenly divide the domain, or randomly pick n+1 values from the domain. However, poorly chosen parameters cause unpredictable results. The following figure shows four data points and three interpolating curves, each of which is obtained using a different set of parameters. One of them bends outward too much and creates an unnecessary bulge. The black curve has a peak and a small bulge. Only the one between these two follows the trend of data points closely. Thus, the choice of parameters affects the shape of the curve, and, consequently, affects the parameterization of the curve.



On the next few pages, we shall discuss a number of parameter selection methods, which include the **Uniformly Spaced Method**, **Chord Length Method** and **Centripetal Method**. Once a set of parameters is obtained, we need to compute a knot vector. See **Knot Vector Generation** for the details. We will also discuss the **Universal Method** in which a uniformly spaced knot vector is chosen and then used for computing the parameters. **Please keep in mind that there are other methods for selecting parameters.**

The Uniformly Spaced Method

The simplest parameter selection method is the **uniformly spaced method**. Suppose the domain, as usual, is [0,1] and n+1 uniformly spaced parameters are required. The first and the last parameters must be 0 and 1 because we want the curve to pass through the first and the last data points. Therefore, we have $t_0 = 0$ and t_n=1.

Since n+1 points divide the interval [0,1] into n subintervals evenly, each of which must be of length 1/n, the division points are 0, 1/n, 2/n, 3/n, ..., (n-1)/n and 1. Thus, we have

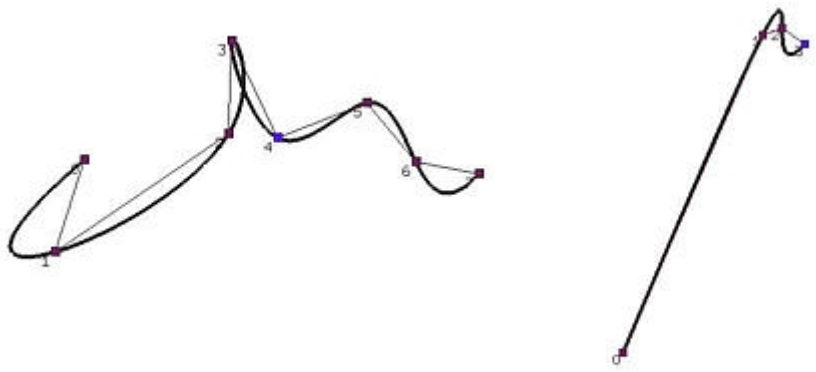
$$\begin{cases} t_0 = 0 \\ t_i = \dfrac{i}{n} \\ t_n = 1 \end{cases} \qquad \text{for } 1 \le i \le n-1$$

For example, if we need 5 parameters, n = 4, and the uniformly spaced parameters are 0, 1/4, 1/2, 3/4 and 1. If we need 8 parameters, n = 7, and the uniformly spaced parameters are 0, 1/7, 2/7, 3/7, 4/7, 5/7, 6/7 and 1.

What if the domain is [a,b] rather than [0,1]? In this case, [a,b] is divided into n intervals by n+1 division point, a and b included. Since the length of this interval is b - a, the length of each subinterval is (b-a)/n. Hence, the division points (*i.e.*, parameters) are

$$\begin{cases} t_0 = a \\ t_i = a + i\dfrac{b-a}{n} \\ t_n = b \end{cases} \qquad \text{for } 1 \le i \le n-1$$

While the uniformly spaced method is simple, it is known to generate some unpleasant results. For example, when data points are not uniformly spaced, using uniformly spaced parameters could generate erratic shapes such as big bulges, sharp peaks and loops. In Figure (a) below, there is a loop at data point 3. In Figure (b), the curve wiggles its way through data points 1, 2 and 3. While we cannot say that these problems are unique to the uniformly spaced method, it does occur more frequently than with other methods.



(a) (b)

The Chord Length Method

If an interpolating curve follows very closely to the data polygon, the length of the curve segment between two adjacent data points would be very close to the length of the *chord* of these two data points, and the length of the interpolating curve would also be very close to the total length of the data polygon. In the figure below, each curve segment of an interpolating polynomial is very close to the length of its supporting chord, and the length of the curve is close to the length of the data polygon. Therefore, if the domain is subdivided according to the distribution of the chord lengths, the parameters will be an approximation of the **arc-length parameterization**. This is the merit of the *chord length* or *chordal* method.

Suppose the data points are $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_n$. The length between $\mathbf{D}_{i-1}$ and $\mathbf{D}_i$ is $|\mathbf{D}_i-\mathbf{D}_{i-1}|$, and the length of the data polygon is the sum of the lengths of these chords:

$$L = \sum_{i=1}^{n} \left| \mathbf{D}_i - \mathbf{D}_{i-1} \right|$$

Therefore, the ratio of the chord length from data point $\mathbf{D}_0$ to data point $\mathbf{D}_k$, denoted as $\boldsymbol{L}_k$, over the length of the data polygon is

$$L_k = \sum_{i=1}^{k} \left| \mathbf{D}_i - \mathbf{D}_{i-1} \right|$$

If we prefer to have an arc-length parameterization of the interpolating curve, the domain has to be divided according to the ratio $\boldsymbol{L}_k$. More precisely, if the domain is [0,1], then parameter t_k should be located at the value of $\boldsymbol{L}_k$:

$$\begin{aligned} t_0 &= 0 \\ t_k &= \frac{\sum_{i=1}^{k} \left| \mathbf{D}_i - \mathbf{D}_{i-1} \right|}{L} \\ t_n &= 1 \end{aligned}$$

where $\boldsymbol{L}$ is the length of the data polygon. In this way, the parameters divide the domain into the ratio of the chord lengths.

Let us look at an example. Suppose we have four data points (n = 3): $\mathbf{D}_0$ = < 0,0 >, $\mathbf{D}_1$ = < 1,2 >, $\mathbf{D}_2$ = < 3,4 > and $\mathbf{D}_3$ = < 4,0 >. The length of each chord is

$$\begin{aligned} \left| \mathbf{D}_1 - \mathbf{D}_0 \right| &= \sqrt{5} = 2.236 \\ \left| \mathbf{D}_2 - \mathbf{D}_1 \right| &= 2\sqrt{2} = 2.828 \\ \left| \mathbf{D}_3 - \mathbf{D}_2 \right| &= \sqrt{17} = 4.123 \end{aligned}$$

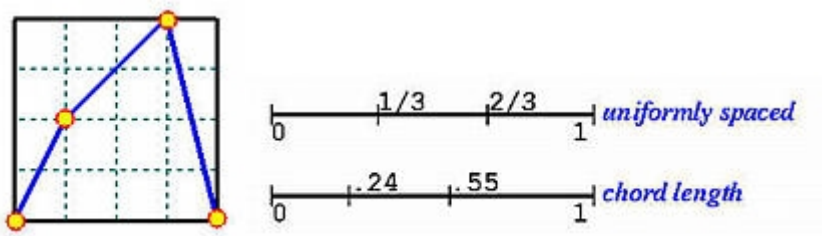
and the total length is

$$L = \sqrt{5} + 2\sqrt{2} + \sqrt{17} = 9.8176$$

Finally, we have the corresponding parameters:

$$t_0 = 0$$
$$t_1 = \frac{|\mathbf{D}_1 - \mathbf{D}_0|}{L} = 0.2434$$
$$t_2 = \frac{|\mathbf{D}_1 - \mathbf{D}_0| + |\mathbf{D}_2 - \mathbf{D}_1|}{L} = 0.5512$$
$$t_3 = 1$$

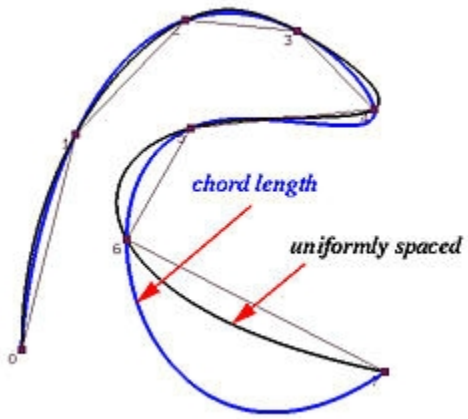
The following figures show the data points and the parameter distributions of the uniformly spaced method and the chord length method.



What if the domain is [a,b] rather than [0,1]? Note that $\boldsymbol{L}_k$ is a ratio between 0 and 1. Since the length of [a, b] is b-a, $\boldsymbol{L}_k(b\text{-}a)$, $0 \le k \le n$, divide [0,b-a] into the same ratio as we did for [0,1]. Therefore, the following parameters divide [a,b] according to the chord lengths:

$$t_0 = a$$
$$t_i = a + L_k(b-a)$$
$$t_n = b$$

The chord length method is widely used and usually performs well. Since it is known (proved by R. Farouki) that polynomial curves cannot be parameterized to have unit speed (*i.e.*, arc-length parameterization), the chord length can only be an approximation. Sometimes, a longer chord may cause its curve segment to have a bulge bigger than necessary. In the figure below, the black and blue curves both interpolate 7 data points. As you can see, both curves have very similar shape, except for the last segment, and the one computed with chord length method wiggling a little. The last curve segments are very different and the curve using the chord length method has a large bulge and twists away from the red curve produced by the uniformly spaced method. This is a commonly seen problem with the chord length method.



The Centripetal Method

E. T. Y. Lee proposed this centripetal method. Suppose we are driving a car through a slalom course. We have to be very careful at sharp turns so that the normal acceleration (*i.e.*, centripetal force) should not be too large. Otherwise, our car may lose control. To drive safely, Lee suggested that the normal force along the path should be proportional to the change in angle. The centripetal method is an approximation to this model. In fact, we can consider the centripetal method as an extension to the chord length method.

Suppose the data points are $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_n$. First of all, we should select a positive "power" value a. Usually, it is $a = 1/2$ for square root. Then, the distance between two adjacent data points is measured by $|\mathbf{D}_k - \mathbf{D}_{k-1}|^a$ rather than the conventional $|\mathbf{D}_k - \mathbf{D}_{k-1}|$. The length of the data polygon under this new measure is

$$L = \sum_{i=1}^{n} \left|\mathbf{D}_i - \mathbf{D}_{i-1}\right|^{\alpha}$$

The ratio of the distance from $\mathbf{D}_0$ to $\mathbf{D}_k$ on the data polygon over the total length is

$$L_k = \frac{\sum_{i=1}^{k} \left|\mathbf{D}_i - \mathbf{D}_{i-1}\right|^{\alpha}}{L}$$

Therefore, $\boldsymbol{L}_0 = 0$, $\boldsymbol{L}_1$, ..., $\boldsymbol{L}_n = 1$ divide [0,1] according to the length of data polygon under the new distance measure. Hence, the parameters are

$$t_0 = 0$$
$$t_k = \frac{\sum_{i=1}^{k} \left|\mathbf{D}_i - \mathbf{D}_{i-1}\right|^{\alpha}}{L}$$
$$t_n = 1$$

If $a = 1$, the centripetal method reduces to the chord length method, and, hence, the former can be considered as an extension to the latter. If $a < 1$, say $a = 1/2$ (*i.e.*, square root), $|\mathbf{D}_k - \mathbf{D}_{k-1}|^a$ is less than $|\mathbf{D}_k - \mathbf{D}_{k-1}|$. Consequently, the impact of a longer chord (*i.e.*, length > 1) on the length of the data polygon is reduced, and the impact of a shorter chord (*i.e.*, length < 1) on the length of the data polygon is increased. Because of this characteristic, Lee claimed that centripetal method handle sharp turns better than the chord length method.

Let us redo the example discussed in the **chord length method**. We have four data points ($n = 3$): $\mathbf{D}_0 = <0,0>$, $\mathbf{D}_1 = <1,2>$, $\mathbf{D}_2 = <3,4>$ and $\mathbf{D}_3 = <4,0>$. Choose $a = 1/2$ and the length of each chord under this new measure is

$$\left|\mathbf{D}_1 - \mathbf{D}_0\right|^{1/2} = \sqrt{\sqrt{5}} = 1.495$$
$$\left|\mathbf{D}_2 - \mathbf{D}_1\right|^{1/2} = \sqrt{2\sqrt{2}} = 1.682$$
$$\left|\mathbf{D}_3 - \mathbf{D}_2\right|^{1/2} = \sqrt{\sqrt{17}} = 2.031$$

The total length is

$$L = \sqrt{\sqrt{5}} + \sqrt{2\sqrt{2}} + \sqrt{\sqrt{17}} = 5.208$$

Therefore, the parameters are

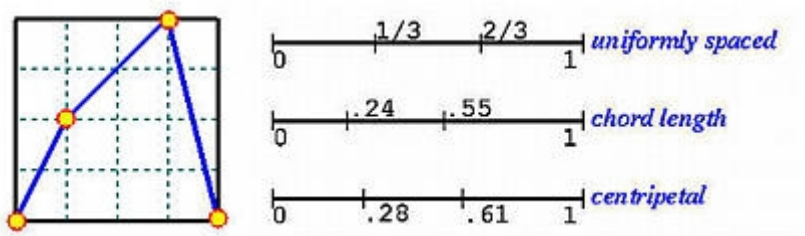
$$t_0 = 0$$

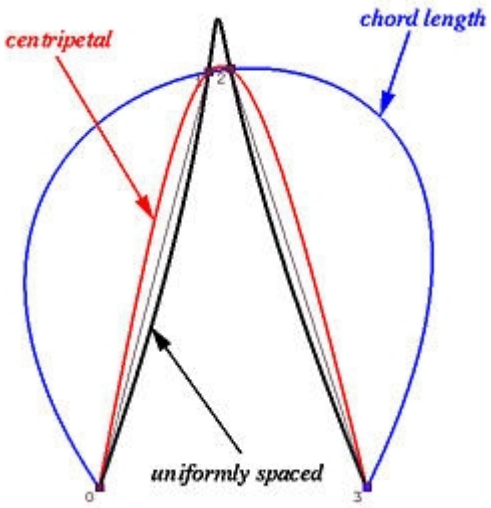
$$t_1 = \frac{|\mathbf{D}_1 - \mathbf{D}_0|^{1/2}}{L} = 0.2871$$

$$t_2 = \frac{|\mathbf{D}_1 - \mathbf{D}_0|^{1/2} + |\mathbf{D}_2 - \mathbf{D}_1|^{1/2}}{L} = 0.6101$$

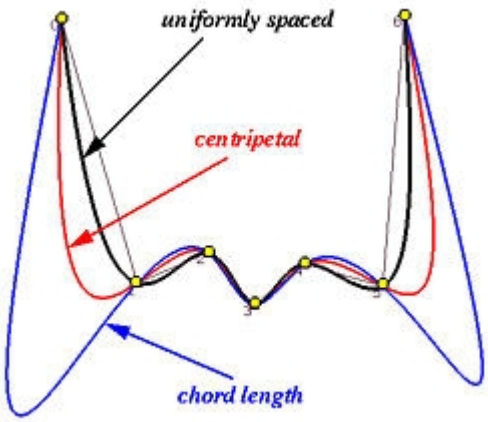
$$t_3 = 1$$

The following gives the distributions of three sets of parameters computed using the uniformly spaced method, the chord length method, and the centripetal method.





Let us take a look at an extreme example. The following shows four data points interpolated by three B-spline curves using the uniformly spaced method (in black), chord length method (in blue) and centripetal method (in red). As you can see, the uniformly spaced method has a peak, the chord length method have two big bulges, and the centripetal method interpolates the two very close adjacent points nicely.



Can we say the centripetal method is superior to the other two? The following example tells a different story. We have 7 data points and the black, blue and red curves are generated using the uniformly spaced method, chord length method and centripetal method. Now, the uniformly spaced method provides a very tight interpolation. The centripetal method is slightly off the the tight result using the uniformly spaced method. Finally, the chord length method wiggles through the two longest chords too much!

Knot Vector Generation

Once a set of parameters is obtained, we can generate a knot vector. Suppose we have n+1 parameters t_0, t_1, ..., t_n and degree p. For a B-spline curve of degree p, we need m+1 knots, where m=n+p+1. If the curve is clamped, these knots are $u_0 = u_1 = ... = u_p = 0$, u_{p+1}, ..., u_{m-p-1}, $u_{m-p} = u_{m-p+1} = ... = u_m = 1$. More precisely, the first p+1 and last p+1 knots are 0's and 1's, respectively. For the remaining n-p knots, they can be uniformly spaced or chosen properly to achieve some desired conditions.

Suppose the remaining n-p internal knots are uniformly spaced. Then, $u_p = 0$, u_{p+1}, ..., u_{m-p-1}, $u_{m-p} = 1$ divide [0,1] into n-p+1 subintervals. Therefore, the knots are

$$u_0 = u_1 = \cdots = u_p = 0$$
$$u_{j+p} = \frac{j}{n-p+1} \qquad \text{for } j = 1, 2, \cdots, n-p$$
$$u_{m-p} = u_{m-p+1} = \cdots = u_m = 1$$

For example, if we have 6 (n = 5) parameters and the degree is p = 3, we should find (n+p+1)+1 = (5+3+1)+1 = 10 knots (*i.e.*, m=9). Since we use clamped curves, the first and the last four (*i.e.*, p+1) knots should be equal to 0 and 1, respectively. More precisely, we have 0, 0, 0, 0, u_4, u_5, 1, 1, 1 and 1, and the two internal knots divide [0,1] into three subintervals, each of which has length 1/3. Hence, the knot vector is { 0, 0, 0, 0, 1/3, 2/3, 1, 1, 1, 1}.

A uniformly spaced knot vector ***does not*** require the knowledge of the parameters, and is very simple to generate. However, this method is ***not*** recommended because, if it is used with the **chord length** method for **global interpolation**, the system of linear equations would be singular.

Another popular method for generating knot vector, suggested by de Boor, is to "average" the parameters. Here is the computation formula:

$$u_0 = u_1 = \cdots = u_p = 0$$
$$u_{j+p} = \frac{1}{p} \sum_{i=j}^{j+p-1} t_i \qquad \text{for } j = 1, 2, \cdots, n-p$$
$$u_{m-p} = u_{m-p+1} = \cdots = u_m = 1$$

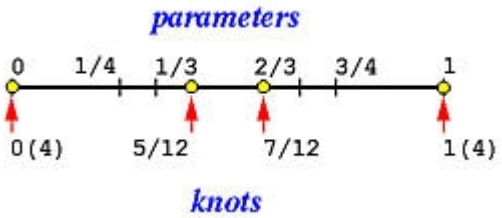
Thus, the first internal knot is the average of p parameters t_1, t_2, ..., t_p; the second internal knot is the average of the next p parameters, t_2, t_3, ..., t_{p+1}. Suppose the 6 (n = 5) parameters are

t_0	t_1	t_2	t_3	t_4	t_5
0	1/4	1/3	2/3	3/4	1

Suppose we need to generate a vector for degree p = 3. As mentioned earlier, 10 knots (m = 9) are required. Of these 10, the first four and last four knots are 0's and 1's, respectively. Thus, the first internal knot is the average of parameters 1/4, 1/3 and 2/3, and the second internal knot is the average of the next three parameters 1/3, 2/3 and 3/4. The computed knot vector is

$u_0 = u_1 = u_2 = u_3$	u_4	u_5	$u_6 = u_7 = u_8 = u_9$
0	(1/4+ 1/3 + 2/3)/3 = 5/12	(1/3 + 2/3 + 3/4)/3 = 7/12	1

The following diagram illustrates the positions of parameters and the generated knots. Note that 0(4) and 1(4) means 0 and 1 are quadruple knots (*i.e.*, multiplicity = 4). As you can see, the first non-zero knot span [0,5/12) contains two parameters, the second non-zero knot span [5/12,7/12) contains no parameter, and the third non-zero knot span [7/12,1) contains two parameters.



The Universal Method

In 1999 Choong-Gyoo Lim proposed an interesting method for selecting parameters. In previously discussed methods, we determine the parameters and then generate a knot vector. Lim's method, known as the **universal method**, works the other way around by using uniformly spaced knots for computing the parameters.

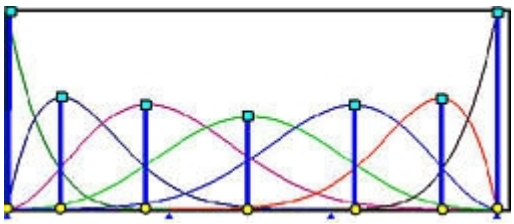
Suppose we need n+1 parameters, one for each data points, and the degree of the desired interpolating B-spline curve is p. Then, the number of knots is m+1, where $m = n + p + 1$. Lim suggested that these knots should be uniformly spaced. More precisely, the first p+1 knots are set to 0, the last p+1 knots are set to 1, and the remaining n - p knots evenly subdivide the domain [0,1]. Therefore, the knots are:

$$u_0 = u_1 = \cdots = u_p = 0$$

$$u_{p+i} = \frac{i}{n-p+1} \qquad \text{for } i = 1, 2, \cdots, n-p$$

$$u_{m-p} = u_{m-p+1} = \cdots = u_m = 1$$

This set of *clamped* type knots defines n+1 B-spline basis functions. Then, the parameters are chosen to be the values at which their corresponding basis functions reach a maximum. Take a look at the following figure, where n = 6 (*i.e.*, 7 data points), p = 4, and m = 11 (*i.e.*, 12 knots). Because we use clamped knots, 0 and 1 are two knots of multiplicity 5 (*i.e.*, p+1) and there are only two internal knots at 1/3 and 2/3. Note that there are n+1 B-spline basis functions, and the maximums of the first and the last are 0 and 1, respectively. The other maximums are marked with blue bars, and the parameters are marked with yellow dots.



Let us do a hand calculation example. Suppose we have 4 data points (*i.e.*, n=3) and degree p=2. Thus, the number of knots is 7 (*i.e.*, $m = n + p + 1 = 6$). Since the knots are uniformly spaced, they are

$u_0 = u_1 = u_2$	u_3	$u_4 = u_5 = u_6$
0	0.5	1

Then, we can compute the desired B-spline Basis functions. Let us start with degree 0 (*i.e.*, $p = 0$). Click **here** to review the definition of B-spline basis functions, and **here** for computation examples. The computed results are:

Knot span	***Basis Function***
$[u_0,u_1) = [0,0)$	$N_{0,0}(u) = 0$

$[u_1,u_2) = [0,0)$	$N_{1,0}(u) = 0$
$[u_2,u_3) = [0,0.5)$	$N_{2,0}(u) = 1$
$[u_3,u_4) = [0.5,1)$	$N_{3,0}(u) = 1$
$[u_4,u_5) = [1,1)$	$N_{4,0}(u) = 0$
$[u_5,u_6) = [1,1)$	$N_{5,0}(u) = 0$

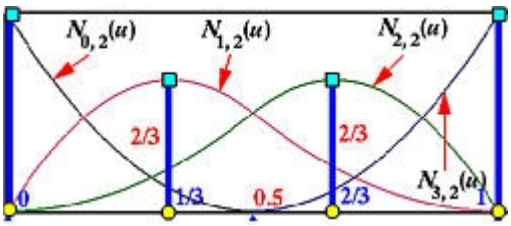
Next, we compute basis functions of degree 1. Since $N_{0,0}(u)$ and $N_{1,0}(u)$ are both 0, $N_{0,1}(u)$ is zero everywhere. Similarly, $N_{4,1}(u)$ is also zero everywhere. Therefore, for basis functions of degree 1, we only need to compute $N_{1,1}(u)$, $N_{2,1}(u)$ and $N_{3,1}(u)$ as shown below:

Basis Function	***Equation***	***Non-zero Range***
$N_{0,1}(u)$	0	everywhere
$N_{1,1}(u)$	$1-2u$	[0,0.5)
$N_{2,1}(u)$	$2u$	[0,0.5)
	$2(1-u)$	[0.5,1)
$N_{3,1}(u)$	$2u-1$	[0.5,1)
$N_{4,1}(u)$	0	everywhere

The basis functions of degree 2 are computed as follows:

Basis Function	***Equation***	***Non-zero Range***
$N_{0,2}(u)$	$(1-2u)^2$	[0,0.5)
$N_{1,2}(u)$	$2u(2-3u)$	[0,0.5)
	$2(1-u)^2$	[0.5,1)
$N_{2,2}(u)$	$2u^2$	[0,0.5)
	$-2(1-4u+3u^2)$	[0.5,1)
$N_{3,2}(u)$	$(2u-1)^2$	[0.5,1)

The following figure shows all four B-spline basis functions of degree 2.



It is not difficult to verify that the maximums of $N_{0,2}(u)$, $N_{1,2}(u)$, $N_{2,2}(u)$ and $N_{3,2}(u)$ are 1 at $u = 0$, 2/3 at $u =$ 1/3, 2/3 at $u = 2/3$ and 1 at $u = 1$. Therefore, with the universal method, the knot vector is { 0, 0, 0, 0.5, 1, 1, 1} and the parameter vector is { 0, 1/3, 2/3, 1}. We have a uniformly spaced parameter in this case. However, the parameters are not necessarily uniformly spaced in general, although the distribution of parameters is usually very even. Because of this fact, the universal method performs like the uniformly spaced method.

Because the knots are uniformly spaced, there is no multiple knots and all basis functions are of C^{p-1} continuous. Click **here** to learn more about continuity of B-spline basis functions. Therefore, in practice, the maximum of a B-spline basis function does not have to be computed precisely. Our experience shows that sampling some values in the non-zero domain and choosing the one with maximum function value usually provides rather accurate result. If accurate maximums are desirable, one-dimensional search techniques such as the Golden Section Search can be used. The search domain is, of course, the non-zero domain of a B-spline basis function.

The universal method has a very interesting property. That is, it is *affine invariant*. This means, the transformed interpolating B-spline curve can be obtained by transforming the data points. This is similar to the affine invariance property of B-spline curves. In fact, it can easily be shown that if the same set of knots and parameters are used in the original and transformed interpolating B-spline curves, then transforming the curve can be achieved by transforming the data points. From this result, we immediately know that the uniformly spaced method is affine invariant, because the knot vector is computed from a set of uniformly spaced parameters which are not changed before and after a transformation. However, the chord length method and centripetal method are ***not*** affine invariant, because in the transformed curve, the distribution of chord lengths may not be the same as the original. As a result, after transforming the data points, we have a new set of chord lengths from which a new set of parameters must be computed.

Parameters and Knot Vectors for Surfaces

Since a B-spline surface of degree (p,q) defined by e+1 rows and f+1 columns of control points has the following equation

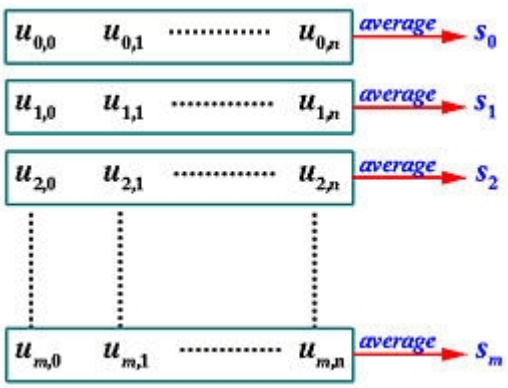
$$\mathbf{S}(u,v) = \sum_{i=0}^{e}\sum_{j=0}^{f}\mathbf{P}_{ij}N_{i,p}(u)N_{j,q}(v)$$

it requires *two* sets of parameters for surface interpolation and approximation. Suppose we have m+1 rows and n+1 columns of data points $\mathbf{D}_{ij}$, where $0 \le i \le m$ and $0 \le j \le n$. Thus, m+1 parameters s_0, ..., s_m in the u-direction (*i.e.*, one for each row of data points) and n+1 parameters t_0, ..., t_n in the v-direction (*i.e.*, one for each column of data points) are required so that point (s_c,t_d) in the domain of the surface corresponds to point $\mathbf{S}(s_c,t_d)$ on the surface, which, in turn, corresponds to the data point $\mathbf{D}_{cd}$. Putting this in an equation form, the point on the surface, evaluated at (s_c,t_d) as shown below,

$$\mathbf{S}(s_c,t_d) = \sum_{i=0}^{e}\sum_{j=0}^{f}\mathbf{P}_{ij}N_{i,p}(s_c)N_{j,q}(t_d)$$

corresponds to the data point $\mathbf{D}_{cd}$, where s_c and t_d are parameters in the u- and v-direction, respectively.

In the equation of the desired B-spline surface, the u-direction corresponds to the index i in $N_{i,p}(u)$ and $\mathbf{P}_{ij}$. Since i runs from 0 to m, $N_{0,p}(u)$, $N_{1,p}(u)$, ..., $N_{m,p}(u)$ are coefficients of the columns of the control points. Therefore, in the u-direction, we need m+1 parameters, and using a parameter computation method that takes degree p and the data points on column j, we can compute m+1 parameters $u_{0,j}$, $u_{1,j}$, ..., $u_{m,j}$. This is shown in the figure below. The desired parameters s_0, s_1, ..., s_m are simply the average of each row. More precisely, parameter s_i is the average of parameters on row i, $s_i = (u_{i,0} + u_{i,1} + ... + u_{i,n})/(n+1)$.



The computation of parameters for the v-direction is similar. Each row of data points has n+1 points and hence requires n+1 parameters. Thus, for data points on row i, we can compute n+1 parameter values $v_{i,0}$, $v_{i,1}$, ..., $v_{i,n}$. Since we have m+1 rows, these values can be organized into a $(m+1)\times(n+1)$ matrix, and parameter t_j is simply the average of the parameters on column j, $t_j = (v_{0,j} + v_{1,j} + ... + v_{m,j})/(m+1)$. In this way, we obtain n+1 parameters for the v-direction.

This procedure is summarized as follows:

Input: $(m+1)\times(n+1)$ data points $\mathbf{D}_{i,j}$;

Output: Parameters in the u-direction $s_0, ..., s_m$

and parameters in the v-direction $t_0, ..., t_n$;

Algorithm:

/* calculate $s_0, ..., s_m$ */

for $j := 0$ **to** n **do** /* column j */

Compute a set of $m+1$ parameters $u_{0,j}, u_{1,j}, ..., u_{m,j}$;

for $i := 0$ **to** m **do**

$s_i = (u_{i,0} + u_{i,1} + ... + u_{i,n})/(n+1)$;

/* parameters for the u-direction are available */

/* calculate $t_0, ..., t_n$ */

for $i := 0$ **to** m **do** /* row i */

Compute a set of $n+1$ parameters $v_{i,0}, v_{i,1}, ..., v_{i,n}$;

for $j := 0$ **to** n **do**

$t_j = (v_{0,j} + v_{1,j} + ... + v_{n,j})/(m+1)$;

/* parameters for the v-direction are available */

Then, parameter values $s_0, s_1, ..., s_m$ and degree p jointly define a knot vector U in the u-direction. Similarly, parameters $t_0, t_1, ..., t_n$ and degree q jointly define a knot vector V in the v-direction. Click **here** to review the way of computing a knot vector from a set of parameters.

Note that the above is only a conceptual algorithm and is inefficient. Once you know what is going on, it is quite easy to convert it to a very efficient algorithm. Note also that this algorithm only works for the uniformly spaced, chord length and centripetal methods. For the universal method, because there is no data points involved, we can apply uniform knots to one row and one column of data points for computing the parameters.

Solving Systems of Linear Equations

Since solving a system of linear equations is a basic skill that will be used for interpolation and approximation, we will briefly discuss a commonly used technique here. In fact, what we will be using is a slightly more general form. Suppose we have a $n \times n$ coefficient matrix **A**, a $n \times h$ "constant" term matrix **B**, and a $n \times h$ unknown matrix **X** defined as follows:

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1h} \\ b_{21} & b_{22} & \cdots & b_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nh} \end{bmatrix}_{n \times h} \qquad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1h} \\ x_{21} & x_{22} & \cdots & x_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nh} \end{bmatrix}_{n \times h} \qquad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}_{n \times n}$$

Suppose further that they satisfy the following relation:

$$\mathbf{B}=\mathbf{A}\cdot\mathbf{X}$$

If **A** and **B** are known, we need a fast method to solve for **X**. One may suggest the following: compute the inverse matrix $\mathbf{A}^{-1}$ of **A** and the solution is simply $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$. While this is a correct way to solve the problem, it is a little overkill. One might also observe the following fact: column *j* of matrix **B** is the product of matrix **A** and column *j* of matrix **X** as shown below:

$$\begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{nj} \end{bmatrix}$$

With this in mind, we can solve column *j* of **X** using **A** and column *j* of **B**. In this way, the given problem is equivalent to solving *h* systems of linear equations. Actually, this is not a good strategy either because in solving for column 1 of **X** matrix **A** will be destroyed, and, as a result, for each column we need to make a copy of matrix **A** before running the linear system solver.

So, we need a method that does not use matrix inversion and does not have to copy matrix **A** over and over. A possible way is the use of the LU decomposition technique.

LU Decomposition

An efficient procedure for solving $\mathbf{B} = \mathbf{A}\cdot\mathbf{X}$ is the LU-decomposition. While other methods such as Gaussian elimination method and Cholesky method can do the job well, this LU-decomposition method can help accelerate the computation.

The LU-decomposition method first "decomposes" matrix **A** into $\mathbf{A} = \mathbf{L}\cdot\mathbf{U}$, where **L** and **U** are lower triangular and upper triangular matrices, respectively. More precisely, if **A** is a $n \times n$ matrix, **L** and **U** are also $n \times n$ matrices with forms like the following:

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

The lower triangular matrix **L** has zeros in all entries *above* its diagonal and the upper triangular matrix **U** has zeros in all entries *below* its diagonal. If the LU-decomposition of $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$ is found, the original equation becomes $\mathbf{B} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{X}$. This equation can be rewritten as $\mathbf{B} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{X})$. Since **L** and **B** are known, solving for $\mathbf{B} = \mathbf{L} \cdot \mathbf{Y}$ gives $\mathbf{Y} = \mathbf{U} \cdot \mathbf{X}$. Then, since **U** and **Y** are known, solving for **X** from $\mathbf{Y} = \mathbf{U} \cdot \mathbf{X}$ yields the desired result. In this way, the original problem of solving for **X** from $\mathbf{B} = \mathbf{A} \cdot \mathbf{X}$ is decomposed into two steps:

1. Solving for **Y** from $\mathbf{B} = \mathbf{L} \cdot \mathbf{Y}$
2. Solving for **X** from $\mathbf{Y} = \mathbf{U} \cdot \mathbf{X}$

Forward Substitution

How easy are these two steps? It turns out to be *very easy*. Consider the first step. Expanding $\mathbf{B} = \mathbf{L} \cdot \mathbf{Y}$ gives

$$\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1h} \\ b_{21} & b_{22} & \cdots & b_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nh} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \bullet \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1h} \\ y_{21} & y_{22} & \cdots & y_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nh} \end{bmatrix}$$

It is not difficult to verify that column *j* of matrix **B** is the product of matrix **A** and column *j* of matrix **Y**. Therefore, we can solve one column of **Y** at a time. This is shown below:

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \bullet \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

This equation is equivalent to the following:

$$\begin{aligned} b_1 &= l_{11} y_1 \\ b_2 &= l_{21} y_1 + l_{22} y_2 \\ \vdots \quad &\vdots \qquad\qquad\qquad\qquad \ddots \\ b_n &= l_{n1} y_1 + l_{n2} y_2 + l_{n3} y_3 + \cdots + l_{nn} y_n \end{aligned}$$

From the above equations, we see that $y_1 = b_1/l_{11}$. Once we have y_1 available, the second equation yields $y_2 = (b_2 - l_{21} y_1)/l_{22}$. Now we have y_1 and y_2, from equation 3, we have $y_3 = (b_3 - (l_{31} y_1 + l_{32} y_2)/l_{33}$. Thus, we compute y_1 from the first equation and substitute it into the second to compute y_2. Once y_1 and y_2 are available, they are substituted into the third equation to solve for y_3. Repeating this process, when we reach equation *i*, we will have $y_1, y_2, \ldots, y_{i-1}$ available. Then, they are substituted into equation *i* to solve for y_i using the following formula:

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{k=1}^{i-1} l_{i,k} y_k \right]$$

Because the values of the y_i's are substituted to solve for the next value of *y*, this process is referred to as *forward substitution*. We can repeat the forward substitution process for each column of **Y** and its corresponding column of **B**. The result is the solution **Y**. The following is an algorithm:

Input: Matrix $\mathbf{B}_{n \times h}$ and a lower triangular matrix $\mathbf{L}_{n \times h}$
Output: Matrix $\mathbf{Y}_{n \times h}$ such that $\mathbf{B} = \mathbf{L} \cdot \mathbf{Y}$ holds.
Algorithm:

/* there are *h* columns */

```
for j := 1 to h do
    /* do the following for each column */
    begin
        /* compute y_1 of the current column */
        y_{1,j} = b_{1,j} / l_{1,1};
        for i := 2 to n do /* process elements on that column */
            begin
                sum := 0; /* solving for y_i of the current column */
                for k := 1 to i-1 do
                    sum := sum + l_{i,k} × y_{k,j};
                y_{i,j} = (b_{i,j} - sum)/l_{i,i};
            end
    end
```

Backward Substitution

After **Y** becomes available, we can solve for **X** from **Y** = **U**·**X**. Expanding this equation and only considering a particular column of **Y** and the corresponding column of **X** yields the following:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \bullet \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

This equation is equivalent to the following:

$$\begin{aligned} y_1 &= u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + \cdots + u_{1n}x_n \\ y_2 &= \cdots \quad u_{22}x_2 + u_{23}x_3 + \cdots + u_{2n}x_n \\ &\vdots \qquad\qquad \ddots \\ y_n &= \qquad\qquad\qquad u_{nn}x_n \end{aligned}$$

Now, x_n is immediately available from equation n, because $x_n = y_n/u_{n,n}$. Once x_n is available, plugging it into equation n-1

$$y_{n-1} = u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n$$

and solving for x_{n-1} yields $x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/ u_{n-1,n-1}$. Now, we have x_n and x_{n-1}. Plugging them into equation n-2

$$y_{n-2} = u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n$$

and solving for x_{n-2} yields $x_{n-2} = [y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_{n-})]/ u_{n-2,n-2}$.

From x_n, x_{n-1} and x_{n-2}, we can solve for x_{n-3} from equation n-3. In general, after x_n, x_{n-1}, ..., x_{i+1} become available, we can solve for x_i from equation i using the following relation:

$$x_i = \frac{1}{u_{ii}}\left[y_i - \sum_{k=i+1}^{n} u_{i,k}x_k \right]$$

Repeat this process until x_1 is computed. Then, all unknown x's are available and the system of linear equations is solved. The following algorithm summarizes this process:

Input: Matrix $\mathbf{Y}_{n\times h}$ and a upper triangular matrix $\mathbf{U}_{n\times h}$
Output: Matrix $\mathbf{X}_{n\times h}$ such that $\mathbf{Y} = \mathbf{U}\cdot\mathbf{X}$ holds.
Algorithm:

```
/* there are h columns */
for j := 1 to h do
    /* do the following for each column */
    begin
        /* compute x_n of the current column */
        x_{n,j} = y_{n,j} / u_{n,n};
        for i := n-1 downto 1 do /* process elements of that column */
            begin
                sum := 0; /* solving for x_i on the current column */
                for k := i+1 to n do
                    sum := sum + u_{i,k} × x_{k,j};
                x_{i,j} = (y_{i,j} - sum)/u_{i,i};
            end
    end
```

This time we work backward, from x_n backward to x_1, and, hence, this process is referred to as *backward substitution*.

LU-decomposition and forward and backward substitutions, including their subroutines/functions, should be available in many numerical method textbooks and mathematical libraries. Check your numerical methods textbook for the details.

Curve Global Interpolation

The simplest method of fitting a set of data points with a B-spline curve is the *global interpolation* method. The meaning of *global* will be clear later on this page.

Suppose we have n+1 data points $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_n$ and wish to fit them with a B-spline curve of degree p, where $p \leq n$ is an input. With the technique discussed in **Parameter Selection and Knot Vector Generation**, we can select a set of parameter values t_0, t_1, ..., t_n. Note that the number of parameters is equal to the number of data points, because parameter t_k corresponds to data point $\mathbf{D}_k$. From these parameters, a knot vector of m + 1 knots is computed where $m = n + p + 1$. So, we have a knot vector and the degree p, the only missing part is a set of n+1 control points. Global interpolation is a simple way of finding these control points.

Global Curve Interpolation

Given a set of n+1 data points, $\mathbf{D}_0$, $\mathbf{D}_2$, ..., $\mathbf{D}_n$ and a degree p, find a B-spline curve of degree p defined by n+1 control points that passes all data points in the given order.

Finding a Solution

Suppose the desired interpolating B-spline curve of degree p is the following:

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i$$

This B-spline curve has n + 1 *unknown* control points. Since parameter t_k corresponds to data point $\mathbf{D}_k$, plugging t_k into the above equation yields the following:

$$\mathbf{D}_k = \mathbf{C}(u_k) = \sum_{i=0}^{n} N_{i,p}(u_k)\mathbf{P}_i \qquad \text{for } 0 \leq k \leq n$$

Because there are n + 1 B-spline basis functions in the above equation ($N_{0,p}(u)$, $N_{1,p}(u)$, $N_{2,p}(u)$, ..., and $N_{n,p}(u)$ and n +1 parameters (t_0, t_1, t_2, .., and t_n), plugging these t_k's into the $N_{i,p}(u)$'s yields $(n+1)^2$ values. Click **here** to learn how to compute these coefficients. These values can be organized into a $(n+1)\times(n+1)$ matrix **N** in which the k-th row contains the values of $N_{0,p}(u)$, $N_{1,p}(u)$, $N_{2,p}(u)$, ..., and $N_{i,p}(u)$ evaluated at t_k as shown below:

$$\mathbf{N} = \begin{bmatrix} N_{0,p}(t_0) & N_{1,p}(t_0) & N_{2,p}(t_0) & \cdots & N_{n,p}(t_0) \\ N_{0,p}(t_1) & N_{1,p}(t_1) & N_{2,p}(t_1) & \cdots & N_{n,p}(t_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ N_{0,p}(t_n) & N_{1,p}(t_n) & N_{2,p}(t_n) & \cdots & N_{n,p}(t_n) \end{bmatrix}$$

Let us also collect vectors $\mathbf{D}_k$ and $\mathbf{P}_i$ into two matrices **D** and **P** as follows:

$$\mathbf{D} = \begin{bmatrix} d_{01} & d_{02} & d_{03} & \cdots & d_{0s} \\ d_{11} & d_{12} & d_{13} & \cdots & d_{1s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & d_{n3} & \cdots & d_{ns} \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} p_{01} & p_{02} & p_{03} & \cdots & p_{0s} \\ p_{11} & p_{12} & p_{13} & \cdots & p_{1s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & p_{n3} & \cdots & p_{ns} \end{bmatrix}$$

Here, we treat data point $\mathbf{D}_k$ as a vector in s-dimensional space (*i.e.*, $\mathbf{D}_k = [\, d_{k1}, ..., d_{ks}]$) and appears on the k-th row of matrix **D**. Similarly, we treat $\mathbf{P}_i$ as a vector in s-dimensional space (*i.e.*, $\mathbf{P}_i = [\, p_{i1}, ..., p_{is}]$) and appears on the i-th row of matrix **P**. Note that in three-dimensional space, we have $s = 3$, and in a plane we have $s = 2$. Note also that **D** and **P** are both $(n+1)\times s$ matrices. Then, it is not difficult to verify that the relation of $\mathbf{d}_k$'s and t_i's is transformed into the following simpler form:

$$\mathbf{D}=\mathbf{N}\cdot\mathbf{P}$$

Since matrix **D** contains the input data points and matrix **N** are obtained by evaluating B-spline basis functions at the given parameters, **D** and **N** both are known and the only unknown is matrix **P**. Since the above is simply a system of linear equations with unknown **P**, solving for **P** yields the control points and the desired B-spline interpolation curve becomes available. Therefore, the interpolation problem is solved!

An Algorithm

You might have something in doubt because matrices **D** and **P** are not column matrices as discussed in your numerical methods and linear algebra textbooks. This actually causes no problem at all. We can solve this system column by column. More precisely, let the i-th column of **D** and the i-the column of **P** be $\mathbf{d}^i$ and $\mathbf{p}^i$, respectively. From the above system of linear equations, we have the following:

$$\mathbf{d}^i=\mathbf{N}\cdot\mathbf{P}^i$$

Then, solving for $\mathbf{p}^i$ from **N** and $\mathbf{d}^i$ gives the i-th column of **P**. Do this for every i in the range of 0 and h, and we will have a complete **P**. Hence, all control points are computed. As you might have noticed, this is very inefficient. Fortunately, many numerical libraries provide us with a linear system solver that is capable of solving the equation $\mathbf{D} = \mathbf{N}\cdot\mathbf{P}$ efficiently. Thus, fitting a B-spline curve to a set of $n+1$ data points is not very difficult. It boils down to the solution of a system of linear equations. The following algorithm summarizes the required steps:

> **Input:** $n+1$ data points $\mathbf{D}_0$, ... $\mathbf{D}_n$ and a degree p
> **Output**: A B-spline curve of degree p that contains all data points in the *given* order
> **Algorithm**:
>
> > Select a method for computing a set of $n+1$ parameters t_0, ..., t_n;
> > As a by-product, we also receive a knot vector U;

for i := 0 **to** n **do**

for j := 0 **to** n **do**

Evaluate $N_{j,p}(t_i)$ into row i and column j of matrix **N**;

/* matrix **N** is available */

for i := 0 **to** n **do**

Place data point $\mathbf{D}_i$ on row i of matrix **D**;

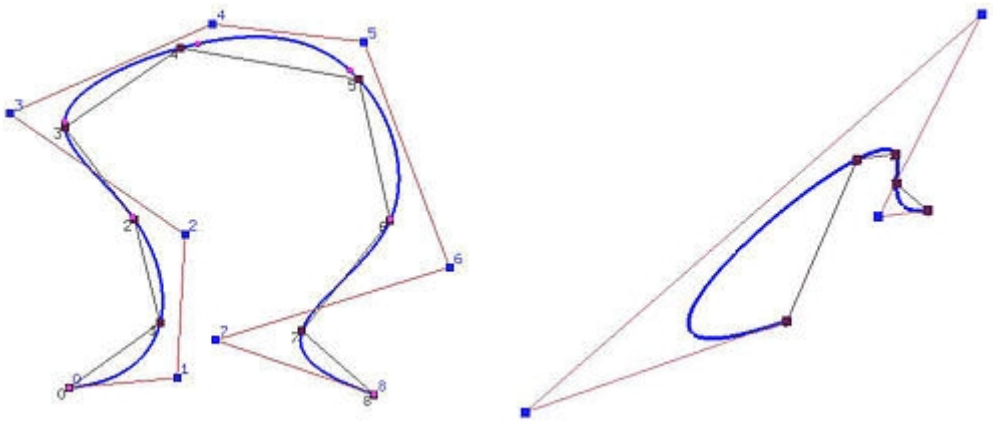
/* matrix **D** is constructed */

Use a linear system solver to solve for **P** from **D** = **N**·**P**

Row i of **P** is control point $\mathbf{P}_i$;

Control points $\mathbf{P}_0$, ..., $\mathbf{P}_n$, knot vector U, and degree p determine an interpolating B-spline curve;

Figure (a) below is an example. There are 9 data points in black. The computed control points are in blue. The small blue dots on the interpolating curve are points corresponding to the knots which are computed using the chord length method. In this case, these "knot points" are quite close to the data points and the control polygon also follows the data polygon closely, although it is not always the case. In Figure (b) below, the data polygon and the control polygon are very different.



(a) **(b)**

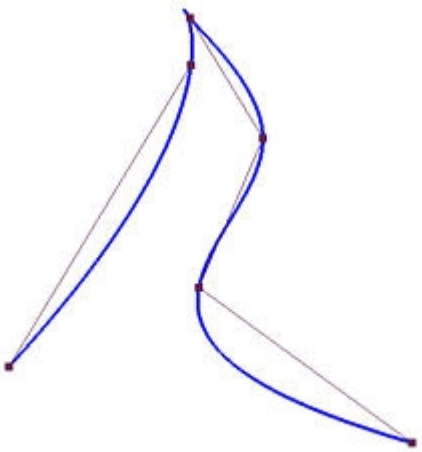
It is important to point out that matrix N is totally positive and banded with a semi-bandwidth less than p (*i.e.*, $N_{i,p}(t_k) = 0$ if $|i - k| \geq p$) if the knots are computed by averaging consecutive p parameters. This was proved by de Boor in 1978. See Knot Vector Generation for the details. This means the system of linear equations obtained from the above algorithm can be solved using Gaussian elimination *without* pivoting.

The Impact of Parameters and Knots

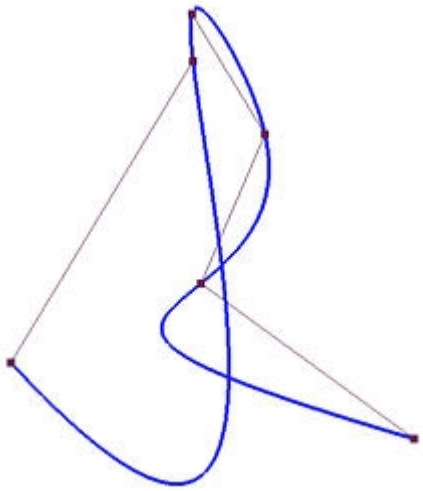
In general, the impact of the selected parameters and knots cannot be predicted easily. However, we can safely say that if the chord length distribution is about the same, all four parameter selection methods should perform similarly. Moreover, the universal method should perform similar to the uniform method because the maximums of B-spline basis functions with uniform knots are distributed quite uniformly. Similarly, the centripetal method should work similar to the chord length method because the former is an

extension to the latter. However, it is not always the case when the distribution of chord lengths change wildly.

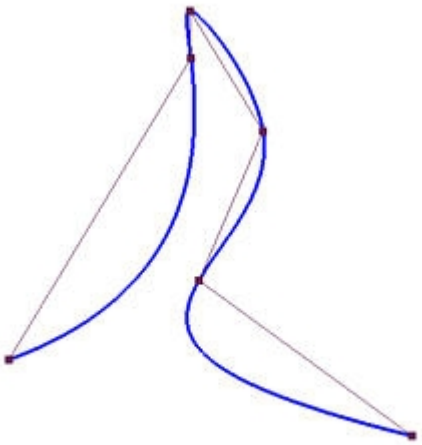
The following four curves are obtained using four parameter selection methods and the degree of the interpolating B-spline curve is 3. The uniform method generates a cusp, the chord length method forces the curve wiggle through data points wildly, the centripetal method resemblances the chord length method but performs better, and the universal method follows the data polygon closely (better than the uniform method) but produces a small loop.



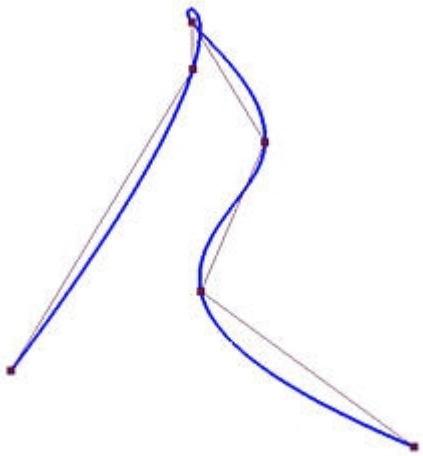
uniform



chord length



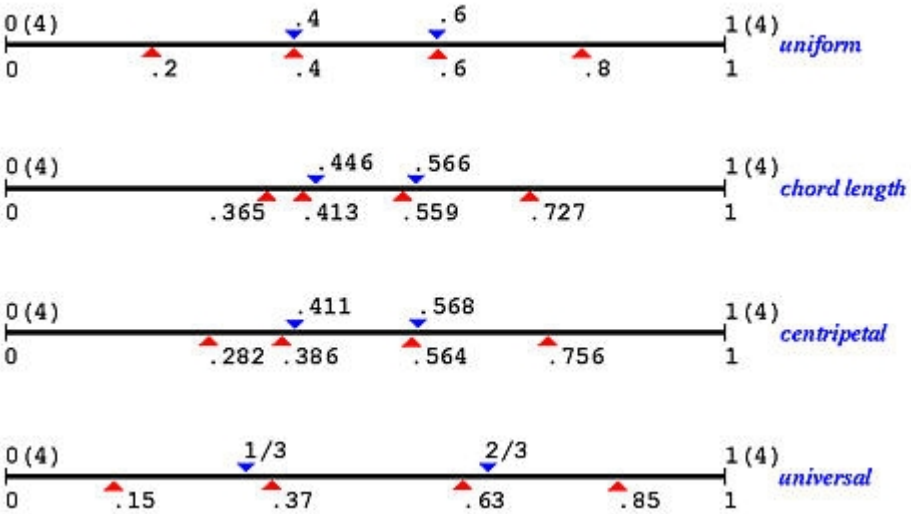
centripetal



universal

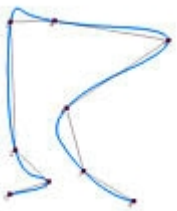
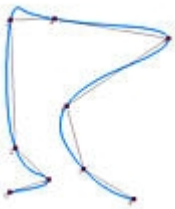
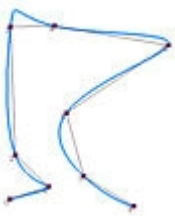
How about the relation between parameters and knots? The following diagram shows the parameters and knots of all four methods. As you can see, the parameters and knots obtained from the universal method is more evenly distributed than those of the chord length method and the centripetal method. Moreover, the parameters and knots obtained from the centripetal method stretch the shorter (*resp*, longer) chords longer (*resp.*, shorter) and hence are more evenly distributed. Therefore, the longer curve segments obtained by the

chord length method become shorter in the centripetal method and the curve does not wiggle wildly through data points.

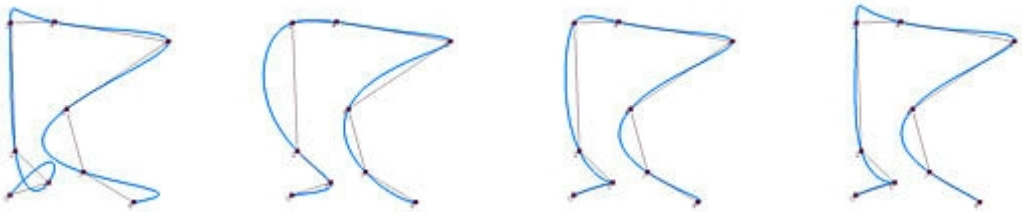


The Impact of Degree

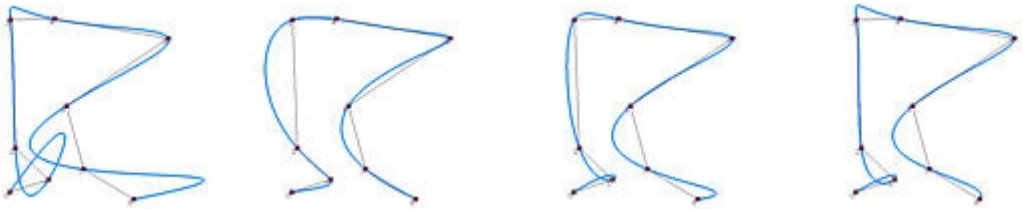
The impact of degree to the shape of the interpolating B-spline curve is also difficult to predict. One can easily observe, from the following images, that the uniformly spaced method and universal method usually follow long chords very well. On the other hand, these two methods have problems with short chords. Because the parameters are equally or almost equally spaced, the interpolating curves have to stretch a little longer for shorter chords. As a result, we see peaks and loops. This situation gets worse with higher degree curves because higher degree curves provide more freedom to wiggle.

	Uniform	**Chord Length**	**Centripetal**	**Universal**
Deg 2				
Deg 3				

Deg 4



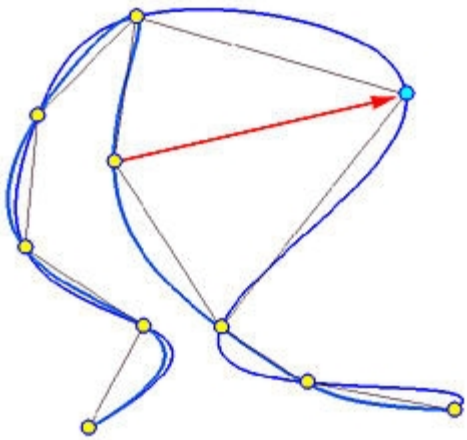
Deg 5



As for the chord length method, the above figures show that it does not work very well for longer chords, especially those followed or preceded by a number of shorter chords, for which big bulges may occur. There is no significant impact of degree on the shape of the interpolating curves shown above. Since the centripetal method is an extension to the chord length method, they share the same characteristics. However, since the centripetal method has a tendency to even out the distance between two adjacent parameters, it also share the same characteristics of the uniform and universal methods. For example, the generated interpolating curves follow longer chords closely and loops may occur for shorter chords when degree increases. In fact, results above show that the centripetal method and the universal method perform similarly.

Why Is This Method Global?

This interpolation method is ***global*** even with the use of B-spline curves which satisfy the local modification property, because changing the position of a single data point changes the shape of the interpolating curve completely. In the following figure, those yellow dots are data points and one of them is moved to its new position, marked in light blue and indicated with a red arrow. These nine data points are fit with a B-spline curve of degree 4 using the centripetal method. As you can see, the resulting curve (in blue) and the original curve have eight data points identical, and the eight curve segments are all different. Therefore, changing the position of s single data point changes the shape of the interpolating curve globally!



Curve Global Approximation

In interpolation, the interpolating curve passes through *all* given data points in the given order. As discussed on the **Global Interpolation** page, an interpolating curve may wiggle through all data points rather than following the data polygon closely. The approximation technique is introduced to overcome this problem by relaxing the strict requirement that the curve *must* contain all data points. Except for the first and last data points, the curve does not have to contain any other point. To measure how well a curve can "approximate" the given data polygon, the concept of *error distance* is used. The error distance is the distance between a data point and its "corresponding" point on the curve. Thus, if the sum of these error distances is minimized, the curve should follow the shape of the data polygon closely. An interpolating curve is certainly such a solution because the error distance of each data point is zero. However, the formulation to be discussed is unlikely to get an interpolating curve. A curve obtained this way is referred to as an approximation curve.

Suppose we are given n+1 data points $\mathbf{D}_0$, $\mathbf{D}_2$, ..., $\mathbf{D}_n$, and wish to find a B-spline curve that can follow the shape of the data polygon without actually containing the data points. To do so, we need two more input: the number of control points (*i.e.*, h+1) and a degree (p), where $n > h \geq p \geq 1$ must hold. Thus, approximation is more flexible than interpolation, because we not only select a degree but also the number of control points. The following is a summary of our problem:

Global Curve Approximation

Given a set of n+1 data points, $\mathbf{D}_0$, $\mathbf{D}_2$, ..., $\mathbf{D}_n$, a degree p, and a number h, where $n > h \geq p \geq 1$, find a B-spline curve of degree p defined by h+1 control points that satisfies the following conditions:

1. **this curve contains the first and last data points (*i.e.*, $\mathbf{D}_0$ and $\mathbf{D}_n$) and**
2. **this curve *approximates* the data polygon in the sense of *least square*.**

With h and p in hand, we can determine a set of parameters and a knot vector. Let the parameters be t_0, t_1, ..., t_n. Note that the number of parameters is equal to the number of data points. Now, suppose the approximation B-spline of degree p is

$$\mathbf{C}(u) = \sum_{i=0}^{h} N_{i,p}(u)\mathbf{P}_i$$

where $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_h$ are the h+1 *unknown* control points. Since we want the curve to pass the first and last data points, we have $\mathbf{D}_0 = \mathbf{C}(0) = \mathbf{P}_0$ and $\mathbf{D}_n = \mathbf{C}(1) = \mathbf{P}_h$. Therefore, there are only h - 1 unknown control points $\mathbf{P}_1$, $\mathbf{P}_2$, ..., $\mathbf{P}_{h-1}$. Taking this into consideration, the curve equation becomes the following:

$$\mathbf{C}(u) = \sum_{i=0}^{h} N_{i,p}(u)\mathbf{P}_i$$

$$\mathbf{C}(u) = N_{0,p}(u)\mathbf{P}_0 + \sum_{i=1}^{h-1} N_{i,p}(u)\mathbf{P}_i + N_{h,p}(u)\mathbf{P}_h$$

The Meaning of Least Square

How do we measure the error distance? Because parameter t_k corresponds to data point $\mathbf{D}_k$, the distance between $\mathbf{D}_k$ and the corresponding point of t_k on the curve is $|\mathbf{D}_k - \mathbf{C}(t_k)|$. Since this distance consists of a square root which is not easy to handle, we choose to use the *squared* distance $|\mathbf{D}_k - \mathbf{C}(t_k)|^2$. Hence, the sum of all *squared* error distances is

$$f(\mathbf{P}_1,...,\mathbf{P}_{h-1}) = \sum_{k=1}^{h-1} \left| D_k - C(t_k) \right|^2$$

Our goal is, of course, to find those control points $\mathbf{P}_1$, ..., $\mathbf{P}_{h-1}$ such that the function $f()$ is minimized. Thus, the approximation is done in the sense of *least square*.

Finding a Solution

This section is going to be a little messy and requires some knowledge in linear algebra. First, let us rewrite $\mathbf{D}_k - \mathbf{C}(t_k)$ into a different form:

$$\begin{aligned} \mathbf{D}_k - C(t_k) &= \mathbf{D}_k - \left[N_{0,p}(u)\mathbf{P}_0 + \sum_{i=1}^{h-1} N_{i,p}(u)\mathbf{P}_i + N_{h,p}(u)\mathbf{P}_h \right] \\ &= \left(\mathbf{D}_k - N_{0,p}(u)\mathbf{P}_0 - N_{h,p}(u)\mathbf{P}_h \right) - \sum_{i=1}^{h-1} N_{i,p}(u)\mathbf{P}_i \end{aligned}$$

In the above, $\mathbf{D}_0$, $\mathbf{D}_k$ and $\mathbf{D}_n$ are given, and $N_{0,p}(t_k)$ and $N_{h,p}(t_k)$ can be obtained by evaluating $N_{0,p}(u)$ and $N_{h,p}(u)$ at t_k. Click **here** to learn how to compute these coefficients. For convenience, let us define a new vector $\mathbf{Q}_k$ as:

$$\mathbf{Q}_k = \left(\mathbf{D}_k - N_{0,p}(u)\mathbf{P}_0 - N_{h,p}(u)\mathbf{P}_h \right)$$

Then, the sum-of-square function $f()$ can be written as follows:

$$f(\mathbf{P}_1,...,\mathbf{P}_{h-1}) = \sum_{k=1}^{h-1} \left| \mathbf{Q}_k - \sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right|^2$$

Next, we shall find out what the squared error distance looks like. Recall the identity $\mathbf{x}\cdot\mathbf{x} = |\mathbf{x}|^2$. This means the inner product of vector $\mathbf{x}$ with itself gives the squared length of $\mathbf{x}$. Thus, the error square term can be rewritten as

$$\begin{aligned} &\left| \mathbf{Q}_k - \sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right|^2 \\ &= \left(\mathbf{Q}_k - \sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \left(\mathbf{Q}_k - \sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \\ &= \mathbf{Q}_k\mathbf{Q}_k - 2\mathbf{Q}_k \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) + \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \end{aligned}$$

Then, function $f()$ becomes

$$f(\mathbf{P}_1,...,\mathbf{P}_{h-1}) = \sum_{k=1}^{h-1} \left[\mathbf{Q}_k\mathbf{Q}_k - 2\mathbf{Q}_k \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) + \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \right) \right]$$

How do we minimize this function? Function $f()$ is actually an elliptic paraboloid in variables $\mathbf{P}_1$, ..., $\mathbf{P}_{h-1}$. Therefore, we can differentiate $f()$ with respect to each $\mathbf{p}_g$ and find the common zeros of these partial derivatives. These zeros are the values at which function $f()$ attends its minimum.

In computing the derivative with respect to $\mathbf{P}_g$, note that all $\mathbf{Q}_k$'s and $N_{i,p}(t_k)$ are constants (*i.e.*, no $\mathbf{P}_k$ involved) and their partial derivatives with respect any $\mathbf{P}_g$ must be zero. Therefore, we have

$$\frac{\partial}{\partial \mathbf{P}_g}\left(\mathbf{Q}_k \cdot \mathbf{Q}_k\right) = 0$$

Consider the second term in the summation, which is the sum of $N_{i,p}(t_k)\mathbf{P}_i \cdot \mathbf{Q}_k$'s. The derivative of each sub-term is computed as follows:

$$\frac{\partial}{\partial \mathbf{P}_g}\left(N_{i,p}(t_k)\mathbf{P}_i \cdot \mathbf{Q}_k\right) = N_{i,p}(t_k)\frac{\partial \mathbf{P}_i}{\partial \mathbf{P}_g} \cdot \mathbf{Q}_k$$

The partial derivative of $\mathbf{P}_i$ with respect to $\mathbf{P}_g$ is non-zero only if $i = g$. Therefore, the partial derivative of the second term is the following:

$$\frac{\partial}{\partial \mathbf{P}_g}\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i \cdot \mathbf{Q}_k\right) = N_{g,p}(t_k) \cdot \mathbf{Q}_k$$

The derivative of the third term is more complicated; but it is still simple. The following uses the multiplication rule (f·g)' = f'·g + f·g'.

$$\frac{\partial}{\partial \mathbf{P}_g}\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right)\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right) = 2\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right)\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\frac{\partial \mathbf{P}_i}{\partial \mathbf{P}_g}\right)$$

Since the partial derivative of $\mathbf{P}_i$ with respect to $\mathbf{P}_g$ is zero if i is not equal to g, the partial derivative of the third term in the summation with respect to $\mathbf{p}_g$ is:

$$\frac{\partial}{\partial \mathbf{P}_g}\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right)\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right) = 2N_{g,p}(t_k)\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right)$$

Combining these results, the partial derivative of $\boldsymbol{f}()$ with respect to $\mathbf{p}_g$ is

$$\frac{\partial f}{\partial \mathbf{P}_g} = -2N_{g,p}(t_k)\mathbf{Q}_k + 2N_{g,p}(t_k)\left(\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i\right)$$

Setting it to zero, we have the following:

$$\sum_{k=1}^{n-1} N_{g,p}(t_k)\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i = \sum_{k=1}^{n-1} N_{g,p}(t_k)\mathbf{Q}_k$$

Since we have h-1 variables, g runs from 1 to h-1 and there are h-1 such equations. Note that these equations are linear in the unknowns $\mathbf{P}_i$'s. What is the coefficient of $\mathbf{P}_i$? Before we go on, let us define three matrices:

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \vdots \\ \mathbf{P}_{h-1} \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} \sum_{k=1}^{n-1} N_{1,p}(t_k)\mathbf{Q}_k \\ \sum_{k=1}^{n-1} N_{2,p}(t_k)\mathbf{Q}_k \\ \vdots \\ \sum_{k=1}^{n-1} N_{h-1,p}(t_k)\mathbf{Q}_k \end{bmatrix} \qquad \mathbf{N} = \begin{bmatrix} N_{1,p}(t_1) & N_{2,p}(t_1) & \cdots & N_{h-1,p}(t_1) \\ N_{1,p}(t_2) & N_{2,p}(t_2) & \cdots & N_{h-1,p}(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ N_{1,p}(t_{n-1}) & N_{2,p}(t_{n-1}) & \cdots & N_{h-1,p}(t_{n-1}) \end{bmatrix}$$

Here, the k-th row of $\mathbf{P}$ is the vector $\mathbf{P}_k$, the k-th row of $\mathbf{Q}$ is the right-hand side of the k-th equation above, and the k-th row of $\mathbf{N}$ contains the values of evaluating $N_{1,p}(u)$, $N_{2,p}(u)$, ..., $N_{h-1,p}(u)$ at t_k. Therefore, if the input data points are s-dimensional vectors, $\mathbf{P}$, $\mathbf{N}$ and $\mathbf{Q}$ are $(h\text{-}1)\times s$, $(n\text{-}1)\times(h\text{-}1)$ and $(h\text{-}1)\times s$ matrices, respectively.

Now, let us rewrite the g-th linear equation

$$\sum_{k=1}^{n-1} N_{g,p}(t_k)\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{P}_i = \sum_{k=1}^{n-1} N_{g,p}(t_k)\mathbf{Q}_k$$

into a different form so that the coefficient of $\mathbf{p}_i$ can easily be read off:

$$\sum_{i=1}^{h-1}\left(\sum_{k=1}^{n-1} N_{g,p}(t_k)N_{i,p}(t_k)\right)\mathbf{P}_i = \sum_{k=1}^{n-1} N_{g,p}(t_k)\mathbf{Q}_k$$

Therefore, the coefficient of $\mathbf{P}_i$ is

$$\sum_{k=1}^{n-1} N_{g,p}(t_k)N_{i,p}(t_k)$$

If you look at matrix $\mathbf{N}$, you should see that $N_{g,p}(t_1)$, $N_{g,p}(t_2)$, ..., $N_{g,p}(t_{n-1})$ are the g-th column of $\mathbf{N}$, and $N_{i,p}(t_1)$, $N_{i,p}(t_2)$, ..., $N_{i,p}(t_{n-1})$ are the i-th column of $\mathbf{N}$. Note that the g-th column of $\mathbf{N}$ is the g-th row of $\mathbf{N}$'s transpose matrix $\mathbf{N}^{\mathrm{T}}$, and the coefficient of $\mathbf{P}_i$ is the "inner" product of the g-th row of $\mathbf{N}^{\mathrm{T}}$ and the i-th column of $\mathbf{N}$. With this observation, the system of linear equations can be rewritten as

$$\left(\mathbf{N}^T\mathbf{N}\right)\mathbf{P} = \mathbf{Q}$$

Since $\mathbf{N}$ and $\mathbf{Q}$ are known, solving this system of linear equations for $\mathbf{P}$ gives us the desired control points. And, of course, we have found the solution.

An Algorithm

Although the development in the previous section is lengthy and tedious, the final result is surprisingly simple: solving a system of linear equations just like we encountered in global interpolation. As a result, the algorithm for global approximation is also quite simple.

Input: n+1 data points $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_n$, a degree p, and the desired number of control points h+1;

Output: A B-spline curve of degree p, defined by h+1 control points, that approximates the given data points;

Algorithm:

Obtain a set of parameters t_0, ..., t_n and a knot vector U;

Let $\mathbf{P}_0 = \mathbf{D}_0$ and $\mathbf{P}_h = \mathbf{D}_n$;

for k := 1 **to** n-1 **do**

Compute $\mathbf{Q}_k$ using the following formula:

$$\mathbf{Q}_k = \left(\mathbf{D}_k - N_{0,p}(u)\mathbf{P}_0 - N_{h,p}(u)\mathbf{P}_h\right)$$

for i := 1 **to** h-1 **do**

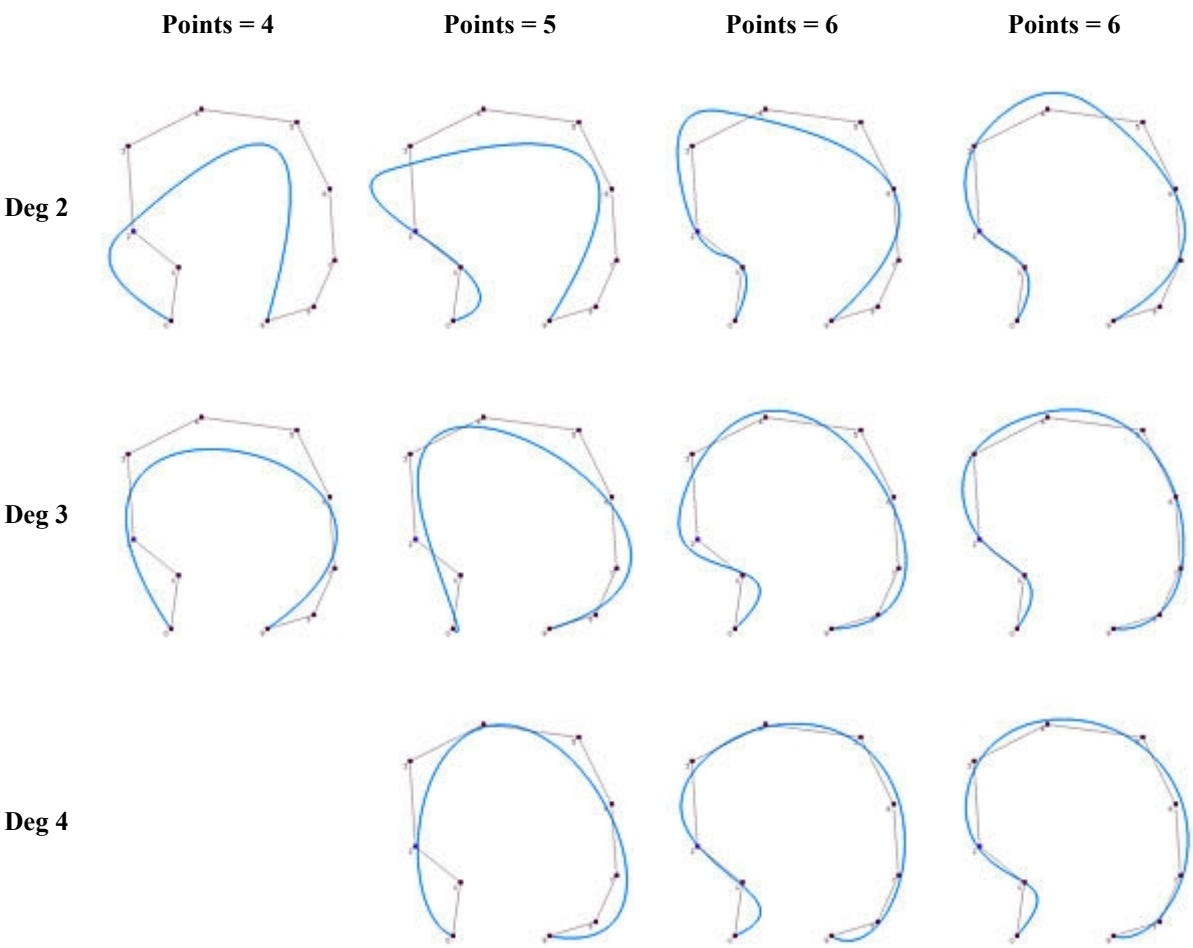
Compute the following and save it to the i-th row of matrix $\mathbf{Q}$;

$$\sum_{i=1}^{h-1} N_{i,p}(t_k)\mathbf{Q}_k$$

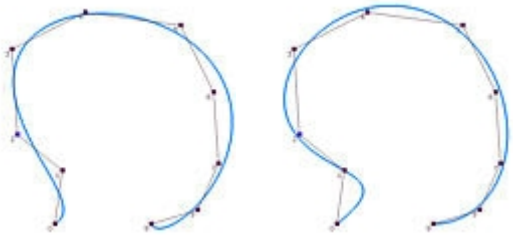
/* matrix **Q** is available */
for k := 1 **to** n-1 **do**
    for i := 1 **to** h-1 **do**
        Compute $N_{i,p}(t_k)$ and save to row k and column i of **N**;
/* matrix **N** is available */
Compute $\mathbf{M} = \mathbf{N}^{T} \cdot \mathbf{N}$;
Solving for **P** from $\mathbf{M} \cdot \mathbf{P} = \mathbf{Q}$;
Row i of **P** is control point $\mathbf{P}_i$;
Control points $\mathbf{P}_0$, ..., $\mathbf{P}_h$, knot vector U and degree p determines an approximation B-spline curve;

The Impact of Degree and Number of Control Points

It is obvious that the data points affect the shape of the approximation curve. What is the impact of degree p and the number of control points on the shape of the curve? The following figures show some approximation curves for 10 data points (n=9) with various degrees and numbers of control points. Each row gives the approximation curves of the same degree but with different number of control points, while each column shows the same number of control points with different degree. Note that all curves use the centripetal method for computing its parameters.



Deg 5

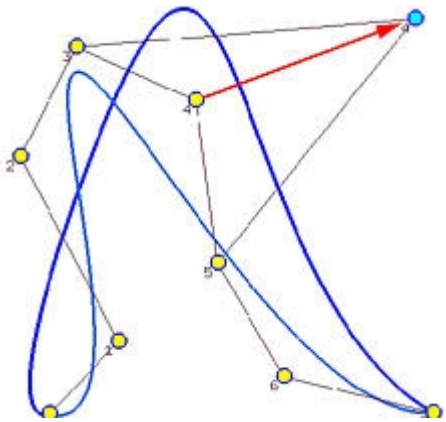


It is understandable that lower degree in general will not be able to approximate the data polygon well because lower degree curves lack of flexibility. Therefore, on each column, a higher the degree yields a better result (*i.e.*, closer to the data polygon). By the same reason, more control points offer higher flexibility of the approximation curve. Hence, on each row, as the number of control points increases, the curve becomes closer to the data polygon.

Shall we use higher degree and many control points? The answer is no because the purpose of using approximation is that we can use fewer number of control points than global interpolation. If the number of control points is equal to the number of data points, we can just use global interpolation! As for degree, we certainly want the smallest one as long as the produced curve can capture the shape of the data polygon.

Why Is This Method Global?

This approximation method is global because changing the position of data point causes the entire curve to change. The yellow dots in the following image are the given data points to be approximated by a B-spline curve of degree 3 and 5 control points (*i.e.*, $n = 7$, $p=3$ and $h = 4$). The parameters are computed with the centripetal method. Suppose data point 4 is moved to a new position indicated by a light blue dot. The new approximation B-spline is the one in blue color. As you can see, except for the first and last data points which are fixed by the algorithm, the shapes of the original curve and the new one are very different. Therefore, changes due to modifying data points are global!



Surface Global Interpolation

Suppose we have m+1 rows and n+1 columns of data points $\mathbf{D}_{ij}$ ($0 \le i \le m$ and $0 \le j \le n$) and wish to find a B-spline surface of degree (p,q) that contains all of them. Similar to the curve case, we have data points and degrees p and q as input. To define an interpolating a B-spline surface, we need two knot vectors U and V, one for each direction, and a set of control points. Like the curve case, the number of control points and the number of data points are equal (*i.e.*, (m+1)×(n+1) control points in m+1 rows and n+1 columns). With the technique discussed in **Parameters and Knot Vectors for Surfaces**, we can compute two sets of parameters s_c ($0 \le c \le m$) in the u-direction and t_d ($0 \le d \le n$) in the v-direction by setting e and f to m and n, respectively. We also obtain knot vectors U and V for the u- and v-direction, respectively. Therefore, what remains to do is to find the desired control points!

Global Surface Interpolation

Given a grid of (m+1)×(n+1) data points $\mathbf{D}_{ij}$ ($0 \le i \le m$ and $0 \le j \le n$) and a degree (p, q), find a B-spline surface of degree (p,q) defined by (m+1)×(n+1) control points that passes all data points in the given order.

Finding a Solution

Suppose the B-spline surface is given as follows:

$$\mathbf{S}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{ij} N_{i,p}(u) N_{j,q}(v)$$

Since it contains all data points and since parameters s_c and t_d correspond to data point $\mathbf{D}_{cd}$, plugging u=s_c and v=t_d into the surface equation yields:

$$\mathbf{D}_{cd} = \mathbf{S}(s_c,t_d) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{ij} N_{i,p}(s_c) N_{j,q}(t_d)$$

Since $N_{i,p}(s_c)$ is independent of the index j, we can move this term out of the summation of j:

$$\begin{aligned}\mathbf{D}_{cd} = \mathbf{S}(s_c,t_d) &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{P}_{ij} N_{i,p}(s_c) N_{j,q}(t_d) \\ &= \sum_{i=0}^{m} N_{i,p}(s_c) \left(\sum_{j=0}^{n} \mathbf{P}_{ij} N_{j,q}(t_d) \right)\end{aligned}$$

If we examine the inner term, we shall see that the index i only appears in $\mathbf{P}_{ij}$. Therefore, we can define the inner expression to be a new term as follows:

$$\mathbf{Q}_{id} = \sum_{j=0}^{n} \mathbf{P}_{ij} N_{j,q}(t_d)$$

More precisely, if i is fixed to the same value, $\mathbf{Q}_{id}$ is the point, evaluated at t_d, on the B-spline curve of degree q defined by n+1 "unknown" control points on row i of the $\mathbf{P}$'s (*i.e.*, $\mathbf{P}_{i0}$, $\mathbf{P}_{i1}$, ..., $\mathbf{P}_{in}$). Plugging $\mathbf{Q}_{id}$ into the equation of $\mathbf{D}_{cd}$ above yields

$$\mathbf{D}_{cd} = \sum_{i=0}^{m} N_{i,p}(s_c)\mathbf{Q}_{id}$$

Thus, data point $\mathbf{D}_{cd}$ is the point, evaluated at s_c, of a B-spline curve of degree p defined by m+1 "unknown" control points on column d of the **Q**'s (*i.e.*, $\mathbf{Q}_{0d}$, $\mathbf{Q}_{1d}$, ..., $\mathbf{Q}_{md}$). Repeating this for every c ($0 \le c \le m$), the d-th column of data points (*i.e.*, $\mathbf{D}_{0d}$, $\mathbf{D}_{1d}$, ..., $\mathbf{D}_{md}$) is obtained from the d-th column of **Q**'s (*i.e.*, $\mathbf{Q}_{0d}$, $\mathbf{Q}_{1d}$, ..., $\mathbf{Q}_{md}$) and parameters s_0, s_1, ..., s_m. Since data points $\mathbf{D}_{0d}$, $\mathbf{D}_{1d}$, ..., $\mathbf{D}_{md}$, degree p and parameters s_0, s_1, ..., s_m are known, this equation means

> **Find the *d*-th column of control points $\mathbf{Q}_{0d}$, $\mathbf{Q}_{1d}$, ..., $\mathbf{Q}_{md}$, given degree *p*, parameters s_0, s_1, ..., s_m and the *d*-th column of data points $\mathbf{D}_{0d}$, $\mathbf{D}_{1d}$, ..., $\mathbf{D}_{md}$.**

So, it is a curve interpolation problem! More precisely, the curve **global interpolation** can be applied to each column of the data points to obtain the columns of control points $\mathbf{Q}_{cd}$'s. Since we have n+1 columns of data points, we shall obtain n+1 columns of **Q**'s.

Now, we can use the same idea but apply to the equation of $\mathbf{Q}_{id}$, which is duplicated below. In this equation, data points on row i of the **Q**'s (*i.e.*, $\mathbf{Q}_{i0}$, $\mathbf{Q}_{i1}$, ..., $\mathbf{Q}_{in}$) are the points on a B-spline curve, evaluated at t_0, t_1, ..., t_n, of degree q defined by n+1 unknown control points $\mathbf{P}_{i0}$, $\mathbf{P}_{i1}$, ..., $\mathbf{P}_{in}$. Therefore, applying curve interpolation with degree q and parameters t_0, t_1, ..., t_n to row i of the **Q**'s gives row i of the desired control points!

$$\mathbf{Q}_{id} = \sum_{i=0}^{m} N_{j,q}(t_d)\mathbf{P}_{ij}$$

Once all rows of control points are found, these control points, along with the two knot vectors and degrees p and q define an interpolating B-spline surface of degree (p,q) of the given data points. Therefore, surface interpolation using B-spline consists of (m+1) + (n+1) curve interpolations! The following summarizes the required steps:

Input: (m+1)×(n+1) data points $\mathbf{D}_{ij}$ and degree (p,q);
Output: A B-spline surface of degree (p,q) that contains all data points;
Algorithm:

Compute parameters in the u-direction s_0, s_1, ..., s_m and the knot vector U;
Compute parameters in the v-direction t_0, t_1, ..., t_n and the knot vector V;
for d := 0 **to** n **do** /* for column d */
    begin /* computing "intermediate data points" **Q**'s */
        Apply curve interpolation to column d of data points (*i.e.*, $\mathbf{D}_{0d}$, $\mathbf{D}_{1d}$, ... $\mathbf{D}_{md}$) using

- degree p
- parameters s_0, s_1, ..., s_m
- knot vector U

        The result is column d of the "intermediate data points" $\mathbf{Q}_{0d}$, $\mathbf{Q}_{1d}$, ..., $\mathbf{Q}_{md}$
    end
for c := 0 **to** m **do** /* for row c */
    begin /* computing the desired control points **P**'s */
        Apply curve interpolation to row c of the **Q**'s (*i.e.*, $\mathbf{Q}_{c0}$, $\mathbf{Q}_{c1}$, ... $\mathbf{Q}_{cn}$) using

- degree q
- parameters $t_0, t_1, ..., t_n$
- knot vector V

The result is row c of the desired control points $\mathbf{P}_{c0}, \mathbf{P}_{c1}, ..., \mathbf{P}_{cn}$

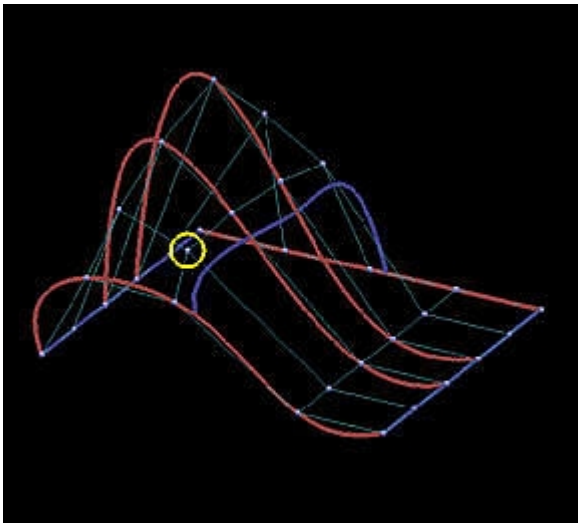
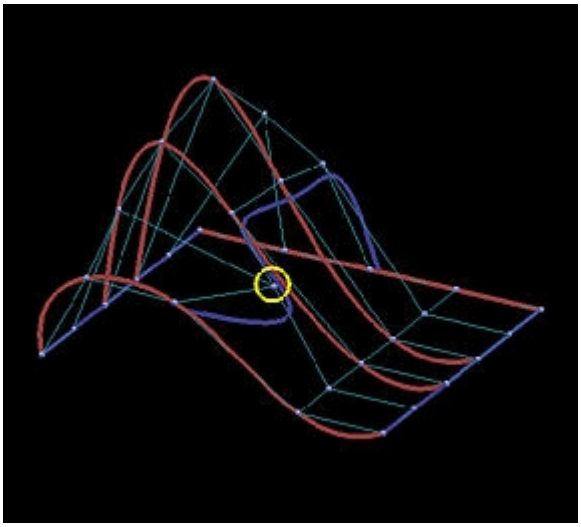
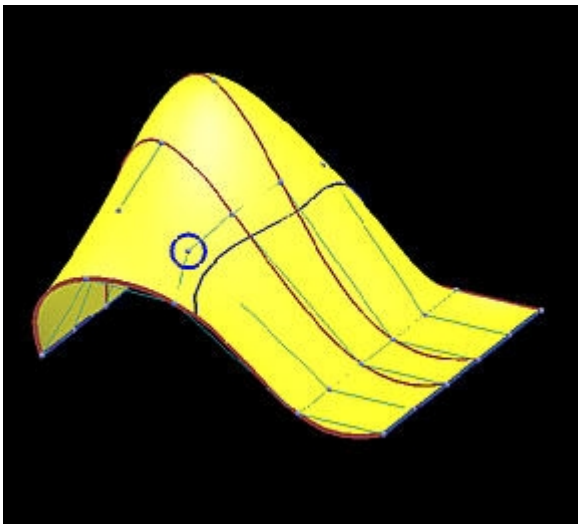
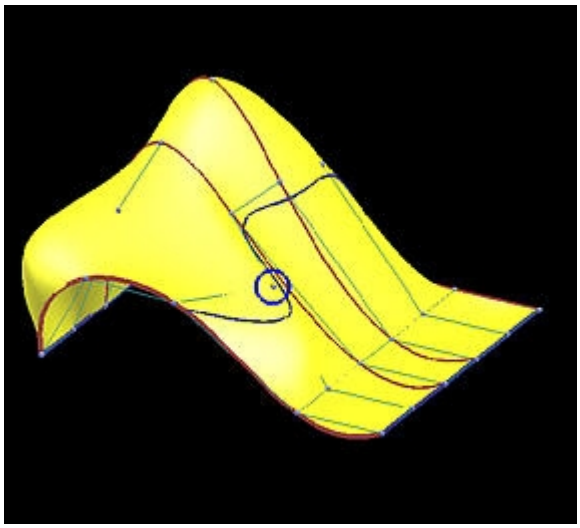
end

The computed $(m+1)\times(n+1)$ control points $\mathbf{P}_{ij}$, degree (p,q), and knot vectors U and V define an interpolating B-spline surface of degree (p,q) for the given data points.

Note that matrix **N** used in the first **for** loop does not change when going from column to column. Therefore, if we naively follow the above algorithm, we would end up solving the system of equation $\mathbf{D} = \mathbf{N}\cdot\mathbf{Q}$ $n+1$ times. This is, of course, not efficient. To speed up, the LU-decomposition of **N** should be computed before the first **for** loop starts, and the interpolation in each iteration is simply a forward substitution followed by a backward substitution. Similarly, a new matrix **N** will be used for the second **for** loop. Its LU-decomposition should also be computed before the second **for** starts. Without doing so, we will perform $(m+1)+(n+1)=m+n+2$ LU-decompositions when only 2 would be sufficient.

Why Is This Method Global?

Because the interpolating surface is obtained through $m+n+2$ global curve interpolations, it is obvious that this interpolation technique is global. The following images show the impact of moving a data point. This B-spline surface is obtained by using six rows (m=5) and five columns (n=4) data points with degree (3,3). Images on the top row show the knot curves (*i.e.*, isoparametric curves that correspond to knots), and the data point marked by a yellow circle indicates the point to be moved. The bottom row shows the corresponding rendered surfaces. It is obvious that after changing the position of a data point, the blue knot curve changes its shape drastically, and moves closer to its neighboring data points. The impact propagates to the right as well because the valley in the far right end is a little deeper as can be seen from the red knot curves. The more dramatic change is shown in the rendered surfaces. Although the indicated data point is moved to the right, this move creates a big bulge in the left end. Please also note that the front boundary curve, actually a red knot curve, also changes its shape slightly. Therefore, the impact of changing the position of a single data point can propagate to entire surface, and, hence, this is a global method.

	Before Move	After Move
Knot Curves		
faces		

Surface Global Approximation

Suppose we wish to find a B-spline surface to approximate the $(m+1)\times(n+1)$ data points. Because an approximation surface does not contain all given data points, like in the curve approximation case, we have controls over the degrees and the number of control points. Therefore, in addition to the data points, the input to this problem includes the degrees p and q in the u- and v-direction, and the number of rows $e+1$ and the number of columns $f+1$ of control points. As in the curve case, the input values must satisfy $m > e \geq p \geq 1$ and $n > f \geq q \geq 1$ to have a solution.

Global Surface Approximation

Given a grid of $(m+1)\times(n+1)$ data points $\mathbf{D}_{ij}$ $(0 \leq i \leq m$ and $0 \leq j \leq n)$ a degree (p, q), and e and f satisfying $m > e \geq p \geq 1$ and $n > f \geq q \geq 1$, find a B-spline surface of degree (p,q) defined by $(e+1)\times(f+1)$ control points $\mathbf{P}_{ij}$ $(0 \leq i \leq e$ and $0 \leq j \leq f)$ that approximates the data point grid in the given order.

Finding a Solution

Let the desired B-spline surface of degree (p,q) defined by the unknown $(e+1)\times(f+1)$ control points $\mathbf{P}_{ij}$'s is the following:

$$\mathbf{S}(u,v) = \sum_{i=0}^{e}\sum_{j=0}^{f}\mathbf{P}_{ij}N_{i,p}(u)N_{j,q}(v)$$

Because there are $m+1$ rows of data points, we need $m+1$ parameters in the u-direction, $s_0, s_1, \ldots, s_m$. Similarly, we need $n+1$ parameters in the v-direction, $t_0, t_1, \ldots, t_n$. The computation of these parameters is discussed in **Parameters and Knot Vectors for Surfaces**. With these parameters in hand, the point on surface that corresponds to data point $\mathbf{D}_{cd}$ is computed as follows:

$$\mathbf{S}(s_c,t_d) = \sum_{i=0}^{e}\sum_{j=0}^{f}\mathbf{P}_{ij}N_{i,p}(s_c)N_{j,q}(t_d)$$

The squared error distance between $\mathbf{D}_{cd}$ and its corresponding point on surface is

$$\left|\mathbf{D}_{cd} - \mathbf{S}(s_c,t_d)\right|^2$$

The sum of all squared error distances is therefore the following:

$$f(\mathbf{P}_{00},\mathbf{P}_{01},\ldots,\mathbf{P}_{ef}) = \sum_{c=0}^{m}\sum_{d=0}^{n}\left|\mathbf{D}_{cd} - \mathbf{S}(s_c,t_d)\right|^2$$

This is a function in the $(e+1)\times(f+1)$ unknown control points $\mathbf{P}_{ij}$'s. Like what we did for the global approximation, to minimize $f()$, we can compute the partial derivatives and set them to zero:

$$\frac{\partial f}{\partial \mathbf{P}_{ij}} = 0$$

Then, we have $(e+1)\times(f+1)$ equations whose common zeros are the desired control points. Unfortunately, these equations are not linear and solving system of non-linear equations is a very time consuming process. Rather than aiming for an optimal solution, we can find a reasonably good solution that does not minimize function $\boldsymbol{f}()$.

To find a non-optimal solution, we shall employ the same technique used in **Global Surface Interpolation**. More precisely, apply curve approximation to each column of data points to compute some "intermediate" data points. In this way, for each column of $m+1$ data points we compute $e+1$ "intermediate" data points. Since there are $n+1$ columns, these "intermediate" data points form a $(e+1)\times(n+1)$ grid. Then, apply curve approximation to each row of these intermediate data points to compute the desired control points. Since each row has $n+1$ "intermediate" data point and there are $e+1$ rows, each approximation for a row generates $f+1$ desired control points, and, as a result, we will finally obtain $(e+1)\times(f+1)$ control points. The following summarizes the required steps:

Input: $(m+1)\times(n+1)$ data points $\mathbf{D}_{ij}$, degree (p,q), and e and f for
$e+1$ rows and $f+1$ columns of control points;
Output: A B-spline surface of degree (p,q) that approximates the data points;
Algorithm:

Compute parameters in the u-direction $s_0, s_1, ..., s_m$ and the knot vector U;
Compute parameters in the v-direction $t_0, t_1, ..., t_n$ and the knot vector V;
for $d := 0$ **to** n **do** /* for column d of **D**'s */
begin /* computing "intermediate data points" **Q**'s */
Apply curve approximation to column d of data points (*i.e.*, $\mathbf{D}_{0d}$, $\mathbf{D}_{1d}$, ... $\mathbf{D}_{md}$) using

- degree p
- parameters $s_0, s_1, ..., s_m$
- knot vector U

The result is column d of the "intermediate data points" $\mathbf{Q}_{0d}, \mathbf{Q}_{1d}, ..., \mathbf{Q}_{ed}$
/* **Q**'s form a $(e+1)\times(n+1)$ matrix */
end
for $c := 0$ **to** e **do** /* for row c of **Q**'s */
begin /* computing the desired control points **P**'s */
Apply curve approximation to row c of the **Q**'s (*i.e.*, $\mathbf{Q}_{c0}$, $\mathbf{Q}_{c1}$, ... $\mathbf{Q}_{cn}$) using

- degree q
- parameters $t_0, t_1, ..., t_n$
- knot vector V

The result is row c of the desired control points $\mathbf{P}_{c0}, \mathbf{P}_{c1}, ..., \mathbf{P}_{cf}$
/* **P**'s form a $(e+1)\times(f+1)$ matrix */
end
The computed $(e+1)\times(f+1)$ control points $\mathbf{P}_{ij}$, degree (p,q), and knot vectors U and V define an approximation B-spline surface of degree (p,q) for the given data points.

In this algorithm, n+1 curve approximations are applied to the columns of **D**'s and then e+1 curve approximations are applied to the rows of the "intermediate" data points. Thus, n+e+2 curve approximations are performed. On the other hand, we could apply m+1 curve approximations to each row of the data points to create (m+1)×(f+1) "intermediate" data points. Then, f+1 curve approximations are applied to each column of this "intermediate" data points to obtain the final (e+1)×(f+1) control points. In this way, m+f+2 curve approximations are performed.

Since this algorithm **does not** minimize function $f()$, it is **not** an optimal one, although it is adequate for many applications.

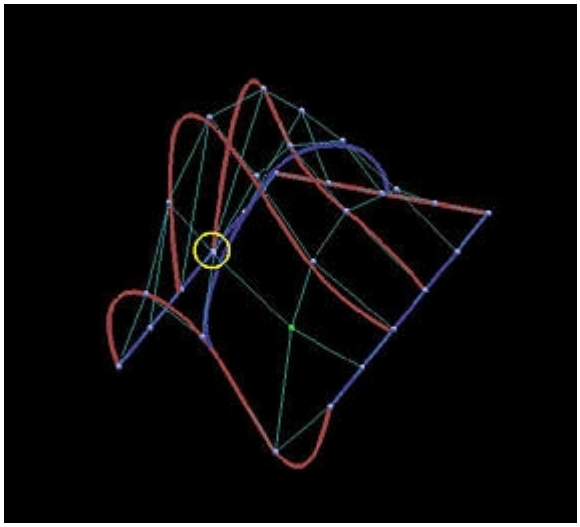
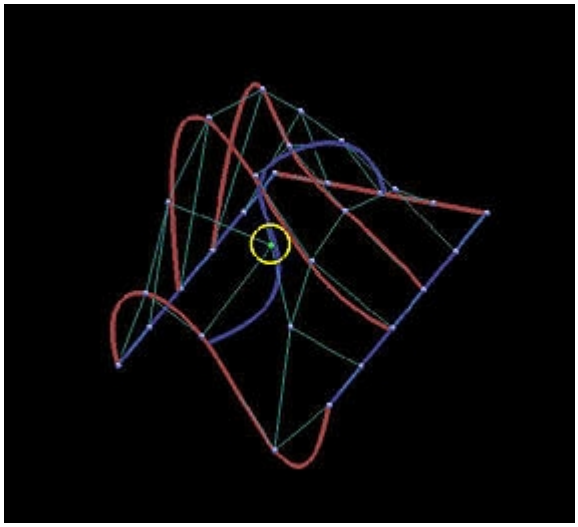
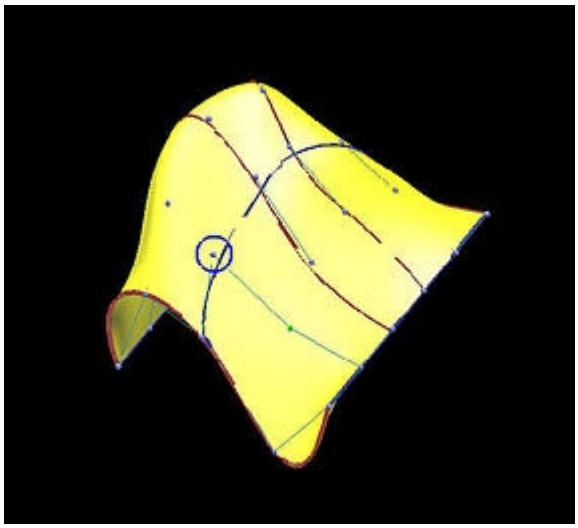
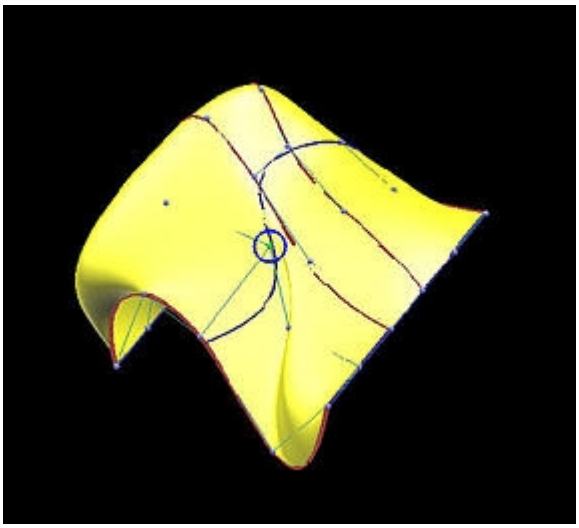
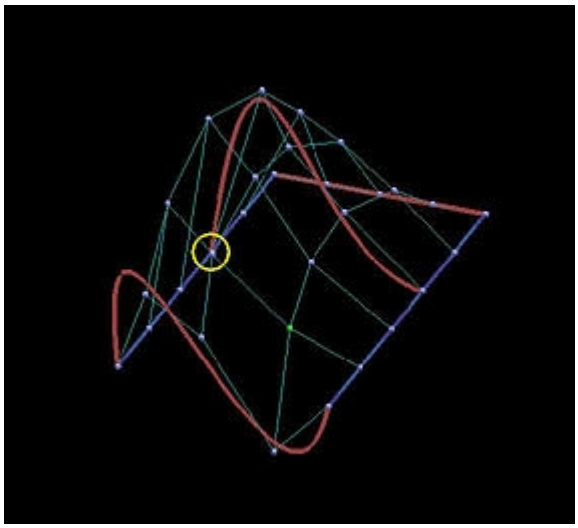
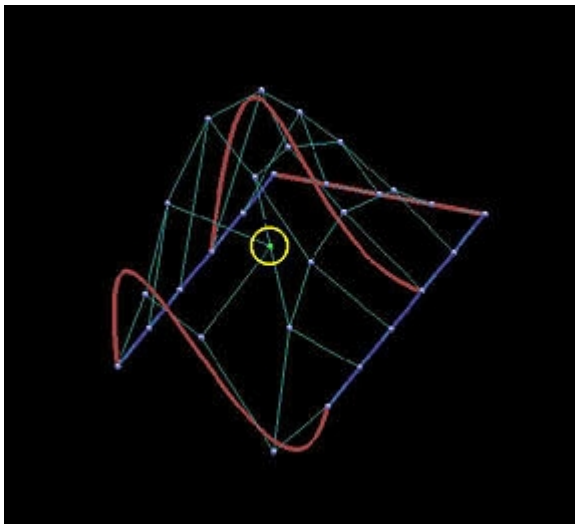
Note that matrix $\mathbf{N}^{\mathbf{T}}\cdot\mathbf{N}$ used in the first **for** loop does not change when going from column to column. Therefore, if we naively follow the above algorithm, we would end up solving the system of equation $\mathbf{Q} = (\mathbf{N}^{\mathbf{T}}\cdot\mathbf{N})\cdot\mathbf{P}$ n+1 times. This is, of course, not efficient. To speed up, the LU-decomposition of $\mathbf{N}^{\mathbf{T}}\cdot\mathbf{N}$ should be computed before the first **for** loop starts, and the approximation in each iteration is simply a forward substitution followed by a backward substitution. Similarly, a new matrix $\mathbf{N}^{\mathbf{T}}\cdot\mathbf{N}$ will be used for the second **for** loop. Its LU-decomposition should also be computed before the second **for** starts. Without doing so, we will perform (e+1)+(n+1)=e+n+2 LU-decompositions when only 2 would be sufficient.

A Simple Comparison

The following is a grid of six rows and five columns. We shall use it to illustrate the differences between interpolation and approximation. Images on the first column are the results before changing the indicated data points, while images on the second depict the results of this move.

The first two rows show the results of global interpolation using the chord length method for determining parameters and knot vectors. The interpolating surface is of degree (3,3). The red and blue curves are knots curves (*i.e.*, isoparametric curves at knots). The indicated data point changes its position slightly, but the blue knot curve changes its shape drastically. The difference between the two interpolating surfaces is even greater. Moving a data point to the right produces a large bulge in the left end of the surface. Moreover, a deep valley below the red knot curve is generated due to this move.

The last two rows show the results of global approximation. The number of control points in u- and v-directions are 5 and 4, respectively. It is obvious that the impact of changing the position of a data point on the shape of the approximation surface is not so dramatic. In fact, in this example, approximation is not so sensitive to changing the position of the indicated data point, and the approximation surfaces follow the shape of the data point net much better than those interpolating surfaces.

	Before Move	After Move
Int: Curve		
Int: Surface		
App: Curve		

App: Surface