

## Lecture 28: Knot Insertion Algorithms for B-Spline Curves and Surfaces

*and the crooked shall be made straight, and the rough ways shall be made smooth;*

Luke 3:5

### 1. Motivation

B-spline methods have several advantages over Bezier techniques. B-splines are piecewise polynomials that meet smoothly at their common boundaries independent of the location of the control points. This guaranteed smoothness allows designers to use low degree polynomial pieces to construct complicated freeform shapes. In addition, a control point of a B-spline curve or surface has no influence on parts of the curve or surface that are far removed from the control point. Thus B-splines provide designers with local control over the shape of a curve or surface.

So far, however, we have no tools for analyzing the geometry of B-spline curves and surfaces. Bezier subdivision facilitates algorithms for rendering and intersecting Bezier curves and expedites procedures for shading and ray tracing Bezier patches. Subdivision also provides a proof of the variation diminishing property for Bezier curves. The goal of this lecture is to develop an analogous tool, called *knot insertion*, for analyzing B-splines curves and surfaces.

Knot insertion is a powerful device with many applications. In this lecture we shall show how to apply knot insertion to convert from B-spline to piecewise Bezier form. We can then use Bezier subdivision to render and intersect the corresponding B-spline curves and surfaces. Alternatively, we shall explain how to apply knot insertion directly to develop fast, robust algorithms for rendering and intersecting B-spline curves and surfaces without converting to piecewise Bezier form. We shall also use knot insertion to prove the variation diminishing property for B-spline curves.

### 2. Knot Insertion

The control points of a B-spline curve describe a piecewise polynomial curve with respect to a fixed set of knots. But any parameter value, even parameter values that represent points internal to a polynomial segment, can be construed as a knot, since the polynomials on either side surely meet smoothly at this join. A knot where the same polynomial appears on both sides is called a *pseudo knot*. Since every spline is a B-spline, we should be able to represent a given spline with respect to a new set of knots, knots that include the original knots as well as a set of pseudo knots. A *knot insertion algorithm* is an algorithm that finds these new B-spline control points from the original B-spline control points, the original knots, and a collection of pseudo knots. Under knot insertion, the control points change, but the curve itself is not affected.

There are many reasons that we shall wish to perform knot insertion. After knot insertion the control polygon of a B-spline curve is closer to the curve than the original control polygon, and in the limit as the distance between the knots approaches zero, the control polygon converges to the B-spline curve. Therefore knot insertion for B-spline curves is similar to subdivision for Bezier curves, so we can use knot insertion as a divide and conquer strategy for analyzing B-spline curves. We can also use knot insertion to convert B-spline curves to piecewise Bezier form. We can then apply Bezier subdivision to render and intersect B-spline curves. In addition, since knot insertion is a corner cutting procedure, we can use knot insertion to prove the variation diminishing property for B-spline curves.

We shall study two different types of knot insertion algorithms: local algorithms and global algorithms. Local knot insertion procedures insert new knots locally, within a single polynomial segment; global knot insertion procedures insert new knots simultaneously in all the polynomial segments of a spline. Local knot insertion procedures include Boehm's algorithm and the Oslo algorithm; global knot insertion procedures include the Lane-Riesenfeld algorithm for uniform B-splines and a new algorithm due to Scott Schaefer for non-uniform B-splines. We will begin with local knot insertion algorithms, and then go on to investigate global knot insertion procedures.

### 3. Local Knot Insertion Algorithms

There are two standard local knot insertion procedures: Boehm's algorithm and the Oslo algorithm. Boehm's algorithm inserts one new knot at a time; the Oslo algorithm computes one new control point at a time. Both Boehm's algorithm and the Oslo algorithm are valid for arbitrary knot sequences and both algorithms can be derived easily from blossoming.

**3.1 Boehm's Knot Insertion Algorithm.** Boehm's knot insertion algorithm inserts one new knot at a time. Consider a cubic B-spline curve with knots  $t_1, \dots, t_m$ . Suppose that we wish to insert a new knot  $u$  between  $t_k$  and  $t_{k+1}$  -- that is, we wish to replace the original knot sequence  $\dots, t_{k-1}, t_k, t_{k+1}, t_{k+2}, \dots$  with the new knot sequence  $\dots, t_{k-1}, t_k, u, t_{k+1}, t_{k+2}, \dots$ . By the dual functional property (Lecture 27, Section 2), the B-spline control points are given by evaluating the blossom of the spline at sequences of consecutive knots. Thus if  $P(t)$  represents the polynomial segment of the spline in the interval  $[t_k, t_{k+1}]$ , then locally we can write both the old and the new control points of the spline in terms of the blossom of  $P(t)$ .

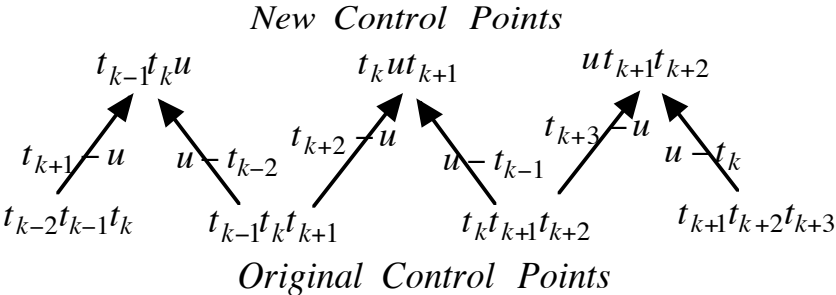
*Original Control Points*

$$\dots, P(t_{k-2}, t_{k-1}, t_k), P(t_{k-1}, t_k, t_{k+1}), P(t_k, t_{k+1}, t_{k+2}), P(t_{k+1}, t_{k+2}, t_{k+3}), \dots$$

*New Control Points*

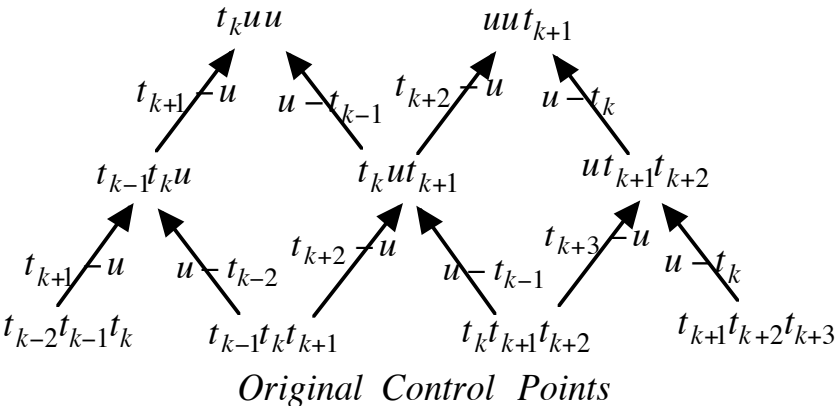
$$\dots, P(t_{k-2}, t_{k-1}, t_k), P(t_{k-1}, t_k, u), P(t_k, u, t_{k+1}), P(u, t_{k+1}, t_{k+2}), P(t_{k+1}, t_{k+2}, t_{k+3}), \dots$$

Our goal is to compute the new control points from the original control points, the original knots, and the one new knot. Since the blossom values of the new control points agree with the blossom values of the original control points in all but one parameter, we can compute the new control points from the original control points using the multi-affine property of the blossom. We illustrate this computation for cubic B-splines in Figure 1.

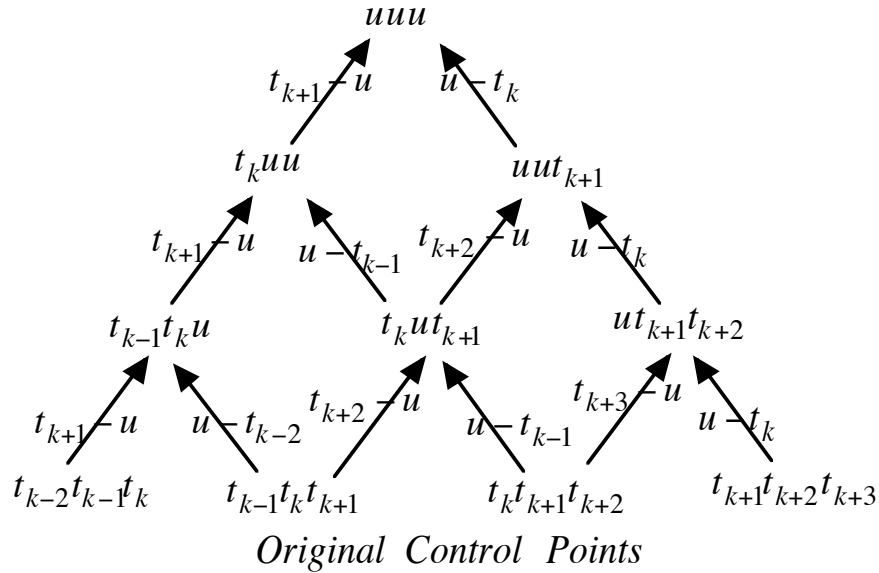


**Figure 1:** Boehm’s algorithm for inserting a new knot  $u$  in the interval  $[t_k, t_{k+1}]$  into a cubic B-spline curve. The original control points are placed at the base of the diagram; the new control points are computed by taking affine combinations of the original control points.

Boehm’s knot insertion algorithm is actually one level of the local de Boor algorithm for the original spline evaluated at the parameter  $u$  (compare Figure 1 with Figure 6 in Lecture 27). If we want to insert the same knot multiple times, we could run Boehm’s algorithm once for each time we wish to insert the knot, but there is a more efficient way. To insert the same knot  $u$  multiple times, we simply run the de Boor algorithm for multiple levels and take the new control points off the left, top, and right edges of the diagram. We illustrate this algorithm for cubic B-splines in Figures 2 and 3. Analogous algorithms hold for splines of arbitrary degree.



**Figure 2:** Boehm’s algorithm for inserting a double knot at  $u$  in the interval  $[t_k, t_{k+1}]$  into a cubic B-spline curve. The original control points are placed at the base of the diagram and the new control points emerge off the left, top, and right edges of the diagram.



**Figure 3:** Boehm’s algorithm for inserting a triple knot at  $u$  in the interval  $[t_k, t_{k+1}]$  into a cubic B-spline curve. The original control points are placed at the base of the diagram and the new control points emerge off the left and right lateral edges of the diagram. Notice that this diagram is the same as the diagram for the de Boor algorithm in the interval  $[t_k, t_{k+1}]$  evaluated at  $t = u$ . Compare to Figure 6 in Lecture 27.

Notice that for a spline of degree  $n$ , inserting a new knot  $n$  times is much like the de Casteljau subdivision algorithm for Bezier curves: the new control points emerge off the left and right lateral edges of the diagram. In fact, the de Casteljau subdivision algorithm for Bezier curves is a special case of Boehm’s  $n$ -fold knot insertion algorithm for B-splines because Bezier curves are B-splines for the knot sequence  $\underbrace{a, \dots, a}_n, \underbrace{b, \dots, b}_n$ .

Boehm’s knot insertion algorithm inserts one new knot at a time. To insert many new knots, simply apply Boehm’s algorithm consecutively multiple times, once for each new knot.

**3.2 The Oslo Algorithm.** Boehm’s knot insertion algorithm inserts one new knot at a time, but computes several new control points simultaneously. The Oslo algorithm inserts many new knots simultaneously in the same interval, but computes only one new control point at a time.

Consider again a cubic B-spline curve with knots  $t_1, \dots, t_m$ . Suppose that we wish to insert the new knots  $u_1, \dots, u_p$  between the old knots  $t_k$  and  $t_{k+1}$  -- that is, we wish to replace the knot sequence  $\dots, t_{k-1}, t_k, t_{k+1}, t_{k+2}, \dots$  with the knot sequence  $\dots, t_{k-1}, t_k, u_1, \dots, u_p, t_{k+1}, t_{k+2}, \dots$ . Let

$P(t)$  represent the polynomial segment of the spline in the interval  $[t_k, t_{k+1}]$ . Then once again by the dual functional property, locally we can write both the old and the new control points of the spline in terms of the blossom of  $P(t)$ .

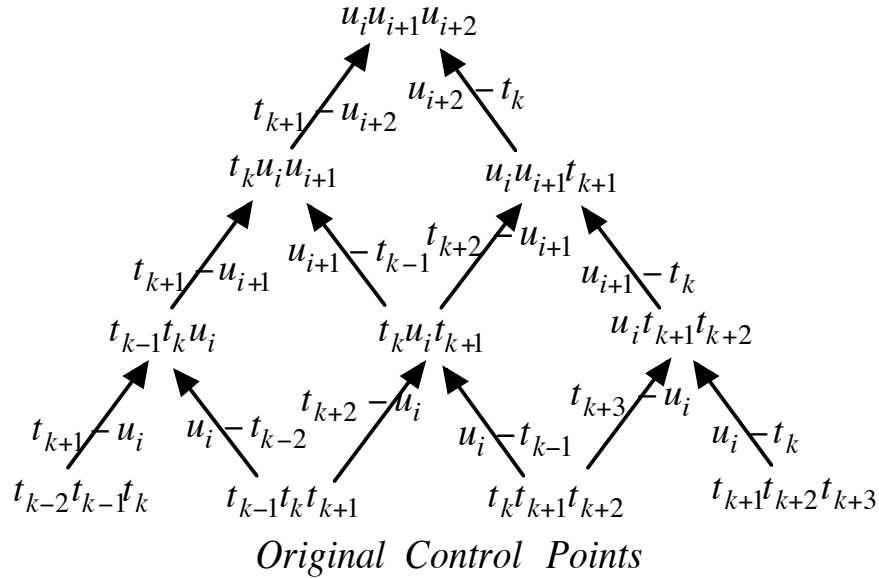
Old Control Points

$$\{p(t_{k+1}, t_{k+2}, t_{k+3})\}$$

New Control Points

$$\{p(u_{i+1}, u_{i+2}, u_{i+3})\}$$

To find the new control points from the original control points, we can simply apply the blossomed version of the de Boor algorithm (see Figure 4). Analogous algorithms apply in arbitrary degree.



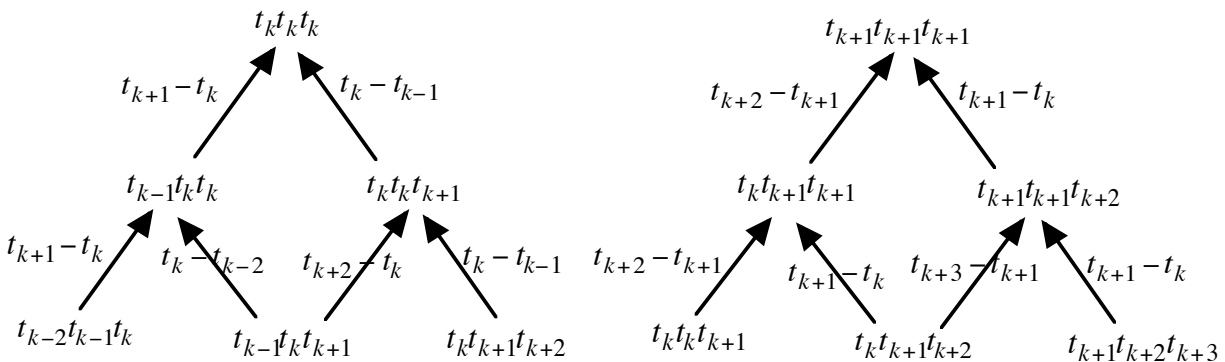
**Figure 4:** The Oslo algorithm for computing a new control point  $u_i u_{i+1} u_{i+2}$  in the interval  $[t_k, t_{k+1}]$  for a cubic B-spline curve. The original control points are placed at the base of the triangle and a new control point emerges at the apex of the triangle. This diagram is just the blossomed version of the de Boor algorithm for the interval  $[t_k, t_{k+1}]$ .

The Oslo algorithm computes only one new control point at a time, so we must run the Oslo algorithm one time for each new control point. Notice, however, that the two initial new control points  $p(t_{k-1}, t_k, u_1)$  and  $p(t_k, u_1, u_2)$  emerge off the left lateral edge of the diagram during the computation of  $p(u_1, u_2, u_3)$ . Similarly, the two final new control points  $p(u_{p-1}, u_p, t_{k+1})$  and  $p(u_p, t_{k+1}, t_{k+2})$  emerge off the right lateral edge of the diagram during the computation of  $p(u_{p-2}, u_{p-1}, u_p)$ .

Many of the standard algorithms for Bezier and B-spline curves can be viewed as particular cases of local knot insertion procedures. Both the de Casteljau evaluation algorithm for Bezier curves and the local de Boor evaluation algorithm for B-splines are special cases of the Oslo algorithm because by the diagonal property of the blossom evaluating a spline of degree  $n$  is the same as inserting a knot of multiplicity  $n$ . In addition, we have already observed that the de Casteljau subdivision algorithm for Bezier curves is just a special case of Boehm's algorithm for inserting a multiple knot into a B-spline curve.

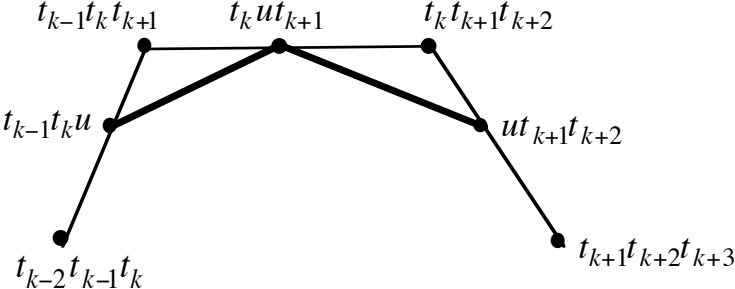
**3.3 Conversion from B-Spline to Piecewise Bezier Form.** Splines are piecewise polynomials. Therefore, one way to analyze B-spline curves and surfaces is to convert to piecewise Bezier form and then apply Bezier subdivision to analyze each of the pieces.

We can apply Boehm's knot insertion algorithm to convert B-spline curves to piecewise Bezier curves. Consider a B-spline curve of degree  $n$  with knots  $t_1, \dots, t_m$ . To convert to piecewise Bezier form, all we need to do is to insure that each knot has multiplicity  $n$ . Since each knot  $t_k$  already has multiplicity at least one, we need only insert each knot at most  $n - 1$  times. We illustrate this procedure in Figure 5 for cubic B-spline curves. Analogous procedures work for B-splines of arbitrary degree.



**Figure 5:** Conversion from B-spline to piecewise Bezier form. Here we illustrate the algorithm for cubic curves and the interval  $[t_k, t_{k+1}]$ . The knots for this B-spline segment are  $t_{k-2}, \dots, t_{k+3}$ ; the knots for the corresponding Bezier segment are  $t_k, t_k, t_k, t_{k+1}, t_{k+1}, t_{k+1}$ . On the left, the knot  $t_k$  is inserted twice starting from the original B-spline control points. On the right the output from the algorithm on the left (in particular the control point  $t_k t_k t_{k+1}$ ) is used to insert the knot  $t_{k+1}$  twice. The Bezier control points emerge along the right lateral edge of the diagram on the left --  $t_k t_k t_k$  and  $t_k t_k t_{k+1}$  -- and along the left lateral edge --  $t_k t_{k+1} t_{k+1}$  and  $t_{k+1} t_{k+1} t_{k+1}$  -- of the diagram on the right.

**3.4 The Variation Diminishing Property for B-Spline Curves.** We can use Boehm’s knot insertion algorithm for converting from B-spline to piecewise Bezier form to prove that B-spline curves are variation diminishing. The key fact we shall need is that Boehm’s knot insertion algorithm is a corner cutting procedure (see Figure 6).



**Figure 6:** A geometric interpretation of Boehm’s knot insertion algorithm for cubic curves. The control points  $t_{k-1}t_k t_{k+1}$  and  $t_k t_{k+1} t_{k+2}$  are replaced by the control points  $t_{k-1}t_k u$ ,  $t_k u t_{k+1}$ , and  $u t_{k+1} t_{k+2}$ , which lie along the line segments of the original control polygon (see Figure 1). Thus Boehm’s knot insertion algorithm is a corner cutting procedure.

**Theorem 3.1:** *B-spline curves satisfy the variation diminishing property.*

Proof: The proof is straightforward and follows directly from the following observations.

- Knot insertion is a corner cutting procedure.
- Therefore, since the Bezier control polygon for a B-spline curve can be generated by knot insertion, the piecewise Bezier control polygon is variation diminishing with respect to the original B-spline control polygon.
- Moreover, each Bezier segment is variation diminishing with respect to its Bezier control polygon.
- Thus the piecewise Bezier curve must be variation diminishing with respect to the original B-spline control polygon.
- But since knot insertion does not alter the underlying curve, the B-spline curve and the piecewise Bezier curve are identical.
- Hence the original B-spline curve must be variation diminishing with respect to the original B-spline control polygon.



**3.5 Algorithms for Rendering and Intersecting B-Spline Curves and Surfaces.** Since we can easily convert from B-spline to piecewise Bezier form, we can use Bezier subdivision to render and intersect B-spline curves and surfaces. We begin with algorithms for B-spline curves.

### *Rendering Algorithm -- B-Spline Curves*

Use Boehm's knot insertion algorithm to convert from B-spline to piecewise Bezier form.

Use de Casteljau's subdivision algorithm to render each of the Bezier segments.

### *Intersection Algorithm -- B-Spline Curves*

Use Boehm's knot insertion algorithm to convert each curve from B-spline to piecewise Bezier form.

Use de Casteljau's subdivision algorithm to intersect each pair of Bezier segments.

Similarly we can use knot insertion to convert tensor product B-spline surfaces into a collection of tensor product Bezier patches. Here we proceed as follows. Recall that to build a tensor product B-spline surface  $B(s,t)$ , we start with a rectangular array of control points  $\{P_{ij}\}$ ,  $i=0,\dots,\mu$ ,  $j=0,\dots,\nu$  and two knot sequences  $s_1,\dots,s_{m+\mu}$  and  $t_1,\dots,t_{n+\nu}$ . We let  $P_i(t)$ ,  $i=0,\dots,\mu$ , be the degree  $n$  B-spline curve with control points  $P_{i,0},\dots,P_{i,\nu}$  and knots  $t_1,\dots,t_{n+\nu}$ , and for each fixed value of  $t$ , we let  $B(s,t)$  be the degree  $m$  B-spline curve for the control points  $P_0(t),\dots,P_\mu(t)$  and knots  $s_1,\dots,s_{m+\mu}$ . Alternatively, we could begin with the B-spline curves  $P_0^*(s),\dots,P_\nu^*(s)$ , where  $P_j^*(s)$  is the degree  $m$  B-spline curve with control points  $P_{0,j},\dots,P_{\mu,j}$  and knots  $s_1,\dots,s_{m+\mu}$ , and for each fixed value of  $s$  let  $B^*(s,t)$  be the B-spline curve for the control points  $P_0^*(s),\dots,P_\nu^*(s)$  and knots  $t_1,\dots,t_{n+\nu}$ . To convert to piecewise Bezier patches, we first apply knot insertion to convert each B-spline curve  $P_0(t),\dots,P_\mu(t)$  into piecewise Bezier form. We then take the resulting curves  $P_0^*(s),\dots,P_\nu^*(s)$  generated by the new knots and the new control points and apply knot insertion to convert these curves into piecewise Bezier form. The resulting array of control points is the control polygon for the piecewise Bezier patches that represent the same surface as the original B-spline surface.

Now to render or ray trace a B-spline surface, we can proceed as follows.

### *Rendering Algorithm -- B-Spline Surfaces*

Use Boehm's knot insertion algorithm to convert from B-spline to piecewise Bezier form.

Use de Casteljau's subdivision algorithm to render each of the Bezier patches.

### *Ray Tracing Algorithm -- B-Spline Surfaces*

Use Boehm's knot insertion algorithm to convert from B-spline to piecewise Bezier form.

Use de Casteljau's subdivision algorithm to ray trace each of the Bezier patches.

Converting from B-spline to piecewise Bezier form solves many problems; we can always render B-spline curves and surfaces by first converting to piecewise Bezier form. From this point of view, B-spline methods are used for design because they have better geometric properties such as guaranteed smoothness, lower degree, and local control. Bezier techniques are used only for analysis, where we can take advantage of Bezier subdivision.

Nevertheless, converting B-splines to piecewise Bezier form for analysis seems a bit extravagant. Piecewise Bezier curves and surfaces have many more control points than the corresponding B-spline curves and surfaces. Surely it would be more efficient to work directly with these B-spline control points, rather than to convert to piecewise Bezier form. Also B-splines lie in the convex hull of their control points. If the convex hulls of two B-spline curves fail to intersect, then the two B-spline curves will not intersect. It seems wasteful to convert both B-spline curves to piecewise Bezier form and then check the convex hulls of every pair of Bezier control polygons. Surely it would be faster to work directly with the B-spline control polygons.

But there is a problem here. When we render Bezier curves and surfaces, we typically subdivide till the curve or surface can be approximated closely by its control polygon or control polyhedron. How should we proceed with B-spline curves and surfaces? If the spline is not approximated closely by its control polygon, we need to insert more knots to get a better approximation. But where should we insert these knots? Similarly, if the convex hulls of two B-spline curves intersect, then we may need to insert more knots to refine our approximation. But again, where should we insert these knots? Boehm's knot insertion algorithm and the Oslo algorithm are local knot insertion procedures. If we do not know where to insert the knots, we cannot effectively apply these procedures. To overcome this difficulty, we shall now turn our attention to global knot insertion algorithms.

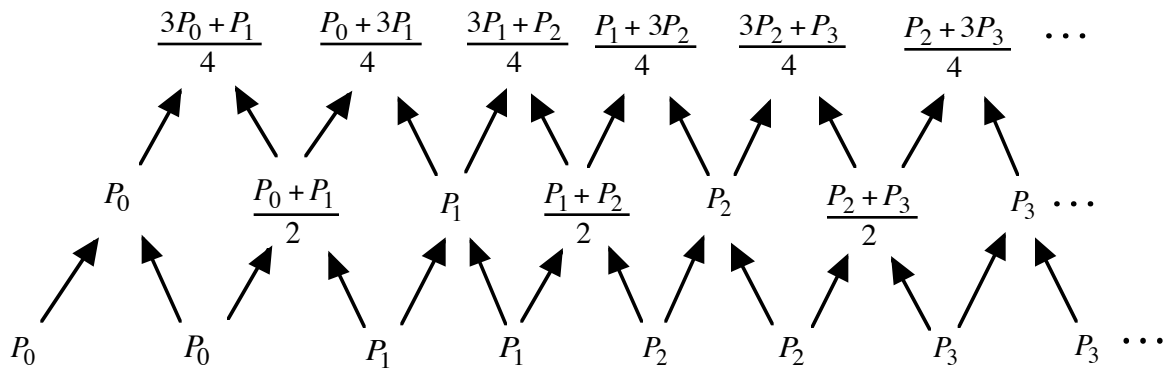
#### **4. Global Knot Insertion Algorithms**

We are going to investigate two global knot insertion procedures: the Lane-Riesenfeld algorithm for uniform B-splines and Schaefer's new algorithm for B-splines with arbitrary knots. Both algorithms insert one new knot between each consecutive pair in the original knots. Iterating these knot insertion algorithms generates control polygons that converge to the original B-spline curve or surface.

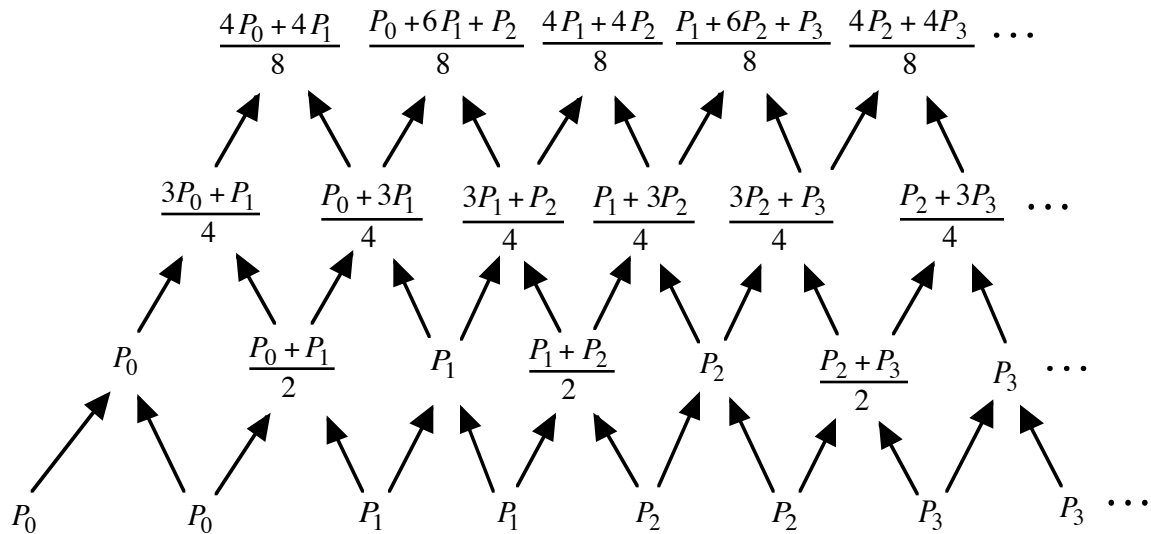
**4.1 The Lane-Riesenfeld Algorithm.** To define a B-spline curve or surface, we need to specify both the control points and the knots. We place the control points in locations that roughly describe some desired shape; if we are unhappy with this shape, we can always adjust the location of the control points. But where should we place the knots? When there is no natural bias in our construction, we usually let the knots be evenly spaced, so typically we place the knots at the

integers. The Lane-Riesenfeld algorithm is a knot insertion procedure that works whenever the knots are evenly spaced.

Given the control points for a spline with uniformly spaced knots, the Lane-Riesenfeld algorithm finds new control points for the same spline where one new knot is inserted midway between each consecutive pair of the original knots. The Lane-Riesenfeld algorithm consists of two basic steps: splitting and averaging. The splitting is done once; the averaging is repeated  $n$  times, where  $n$  is the degree of the spline. We illustrate the Lane-Riesenfeld algorithm for quadratic and cubic splines in Figures 7 and 8.

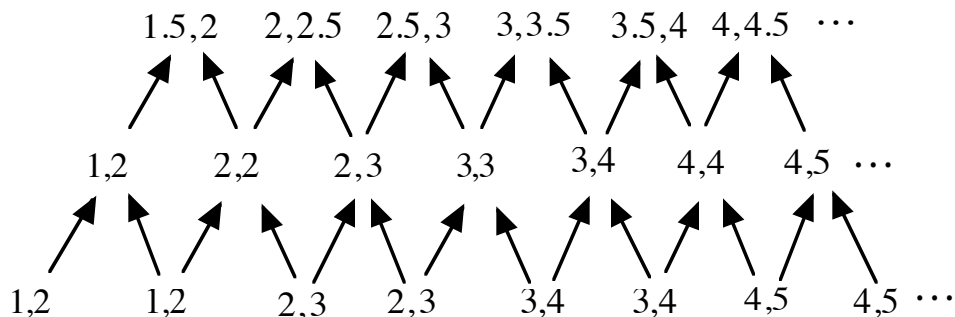


**Figure 7:** The Lane-Riesenfeld algorithm for quadratic B-spline curves. At the bottom of the diagram each control point is split into a pair of points. Adjacent points are then averaged and this averaging step is repeated twice. The new control points for the spline with one new knot inserted midway between each consecutive pair of the original knots emerge at the top of the diagram.

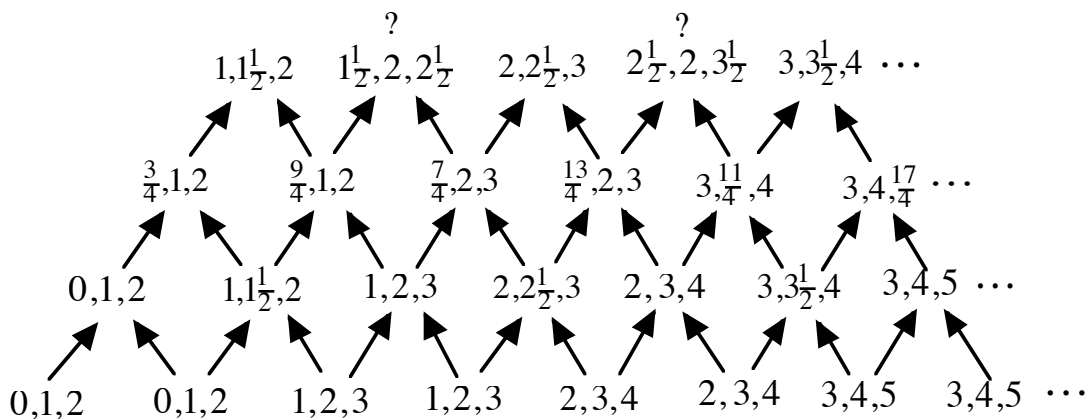


**Figure 8:** The Lane-Riesenfeld algorithm for cubic B-spline curves. At the bottom of the diagram each control point is split into a pair of points. Adjacent points are then averaged and this averaging step is repeated three times. The new control points for the cubic spline with one new knot inserted midway between each consecutive pair of the original knots emerge at the top of the diagram.

We can apply blossoming to establish the validity of the Lane-Riesenfeld algorithm for low degree splines. Figures 9 and 10 illustrate the blossoming interpretation of the Lane-Riesenfeld algorithm for quadratic and cubic B-splines.



**Figure 9:** The blossoming interpretation of the Lane-Riesenfeld algorithm for quadratic B-splines. Here we adopt the multiplicative notation  $uv$  for the blossom  $s(u,v)$ .



**Figure 10:** The blossoming interpretation of the Lane-Riesenfeld algorithm for cubic B-splines. Again we adopt the multiplicative notation  $uvw$  for the blossom  $s(u,v,w)$ .

For quadratic B-splines we can see directly from Figure 9 that the Lane-Riesenfeld algorithm follows easily from the dual functional property together with the symmetry and the multiaffine properties of the blossom. For cubic B-splines, however, the derivation is not so straightforward; blossom values on the top row of Figure 10 above which we have placed the symbol ? are not an immediate consequence of the multiaffine property of the blossom. From the multiaffine property, for the spline  $S(t)$

$$1\frac{1}{2} = \frac{1}{2}\left(\frac{3}{4} + \frac{9}{4}\right) \Rightarrow s(1,1\frac{1}{2},2) = \frac{1}{2}s(\frac{3}{4},1,2) + \frac{1}{2}s(\frac{9}{4},1,2),$$

but why is

$$s(1\frac{1}{2},2,2\frac{1}{2}) = \frac{1}{2}s(\frac{9}{4},1,2) + \frac{1}{2}s(\frac{7}{4},2,3)?$$

To establish this result, observe that by the multiaffine property

$$s\left(\frac{9}{4}, 1, 2\right) = \frac{s(1, 1\frac{1}{2}, 2) + s(1, 2, 3)}{2}$$

$$s\left(\frac{7}{4}, 2, 3\right) = \frac{s(1, 2, 3) + s(2, 2\frac{1}{2}, 3)}{2}.$$

Therefore again by the multiaffine property

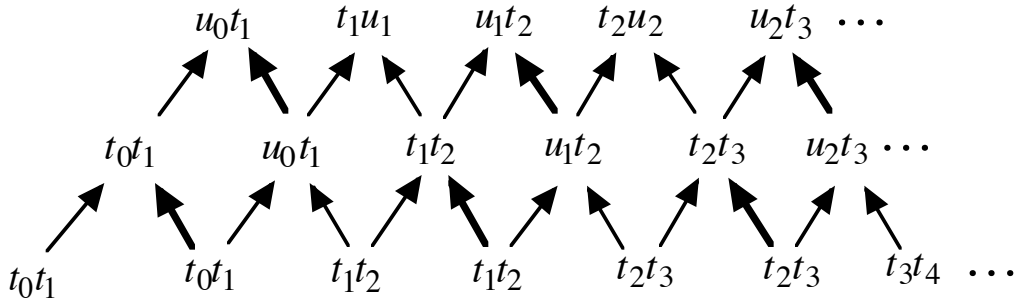
$$\begin{aligned} \frac{1}{2}s\left(\frac{9}{4}, 1, 2\right) + \frac{1}{2}s\left(\frac{7}{4}, 2, 3\right) &= \frac{1}{2}\left(\frac{s(1, 1\frac{1}{2}, 2) + s(1, 2, 3)}{2}\right) + \frac{1}{2}\left(\frac{s(1, 2, 3) + s(2, 2\frac{1}{2}, 3)}{2}\right) \\ &= \frac{1}{4}s(1, 1\frac{1}{2}, 2) + \frac{3}{4}\left(\frac{2}{3}s(1, 2, 3) + \frac{1}{3}s(2, 2\frac{1}{2}, 3)\right) \\ &= \frac{1}{4}s(1, 1\frac{1}{2}, 2) + \frac{3}{4}s(1\frac{1}{2}, 2, 3) = s(1\frac{1}{2}, 2, 2\frac{1}{2}). \end{aligned}$$

Similar arguments hold for all the other blossom values above which we have placed the symbol ?.

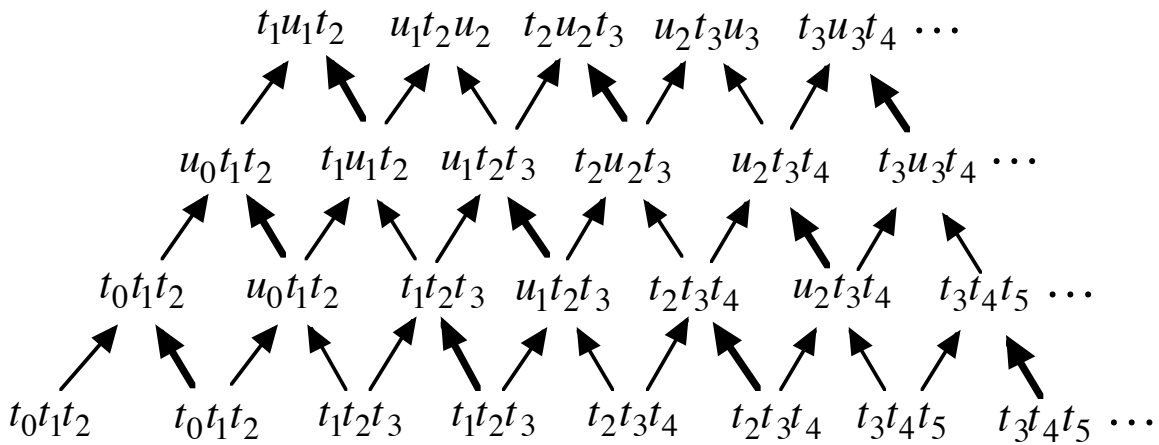
The Lane-Riesenfeld algorithm works not only for knots at the integers, but also whenever the knots are evenly spaced. Indeed if we divide each knot in Figure 9 or 10 by 2 or 4 or  $2^n$ , the algorithm remains valid. Thus we can iterate the Lane-Riesenfeld algorithm by taking the output of one stage as the input for the next stage. We shall see in Section 4.3 that the control polygons generated in this manner converge to the B-spline curve for the original control polygon. Thus we can use the Lane-Riesenfeld algorithm to render and intersect B-spline curves with uniformly spaced knots (see Section 4.4).

Although the Lane-Riesenfeld algorithm is valid in every degree, the blossoming proof gets more and more difficult as the degree increases. The standard proof of the Lane-Riesenfeld algorithm is not based on blossoming, but rather on continuous convolution of B-spline basis functions. This proof is well beyond the scope of this lecture, so we shall not pursue this proof here. Instead, we shall introduce a new global knot insertion algorithm which is not only compatible with blossoming but also works for arbitrary knots.

**4.2 Schaefer's Knot Insertion Algorithm.** Consider a B-spline curve with knots  $t_0, \dots, t_m$ . The goal of Schaefer's algorithm is to insert one new knot  $u_i$  between each consecutive pair  $t_i, t_{i+1}$  of the original knots. As in the Lane-Riesenfeld algorithm, Schaefer's algorithm consists of two basic steps: splitting and averaging. The splitting is done once; the averaging is repeated  $n$  times, where  $n$  is the degree of the spline. The only difference here is that the averaging is not simply the arithmetic mean, adding and dividing by two, but rather is a weighted averaging, where the weights depend on the spacing between the knots. We illustrate Schaefer's algorithm using blossoming for quadratic and cubic B-splines in Figures 11 and 12.



**Figure 11:** Schaefer's knot insertion algorithm for quadratic B-splines. As usual we adopt the multiplicative notation  $uv$  for the blossom  $s(u,v)$ .



**Figure 12:** Schaefer's knot insertion algorithm for cubic B-splines. Again we adopt the multiplicative notation  $uvw$  for the blossom  $s(u,v,w)$ .

To understand how Schaefer's algorithm works in the general case, consider first Schaefer's algorithm for quadratic B-splines illustrated in Figure 11. Notice that at each stage of this algorithm, every other point is promoted to the next stage with no additional computation (indicated by thicker arrows). For example, the control points represented by the blossom values  $u_0t_1$  and  $u_1t_2$  already appear at the first stage and are simply promoted to their upper left into the second stage; therefore the labels on the accompanying arrows are just zero and one. Every other small triangle in the algorithm introduces one new knot using the multiaffine property of the blossom.

To get from the quadratic algorithm to the cubic algorithm, just append the next original knot value to each of the blossoms on the first two stages. For example, on the first stage change  $u_0t_1$  to  $u_0t_1t_2$ . As in the quadratic algorithm, at each stage of the cubic algorithm, every other point is promoted to the next stage with no additional computation, and every other small triangle simply introduces one new knot using the multiaffine property of the blossom.

You can get from the cubic to the quartic version of Schaefer's algorithm in much the same way: simply append the next knot value from the original knots to each of the blossoms in the first three stages. Every other point in the third stage is promoted to the fourth stage with no additional computation, and every other small triangle simply introduces one new knot using the multiaffine property of the blossom. That's it. It's easy. Better still, the proof that this approach works is immediate from the multiaffine and dual functional properties of the blossom. Unlike the Lane-Riesenfeld algorithm, Schaefer's algorithm is highly compatible with blossoming. In fact, Schaefer's algorithm is generated directly from blossoming.

Notice that both Schaefer and Lane-Riesenfeld build their algorithms for inserting knots into B-splines of degree  $n+1$  from their algorithms for inserting knots into B-splines of degree  $n$ . Lane-Riesenfeld simply append one more round of averaging on top of the  $n$  rounds that appear in their algorithm for B-splines of degree  $n$ . Nothing changes in the first  $n$  rounds. On the other hand, the first  $n$  rounds of Schaefer's algorithm for inserting knots into B-splines of degree  $n+1$  are not identical to the  $n$  rounds of Schaefer's algorithm for inserting knots into B-splines of degree  $n$ . The same knots are inserted, but unlike the Lane-Riesenfeld algorithm the (weighted) averages in each round change with the degree.

When the knots are evenly spaced, Schaefer's algorithm is not identical to the Lane-Riesenfeld algorithm. We can see this difference even in the quadratic case. In the Lane-Riesenfeld algorithm the labels on the arrows in the first stage, indeed on every stage, are everywhere  $1/2$ ; in Schaefer's algorithm the labels in the first stage for evenly spaced knots are  $1/4$  and  $3/4$ . Notice, however, how efficient Schaefer's algorithm is even as compared to the Lane-Riesenfeld algorithm. In Schaefer's algorithm half the computations simply promote a value to the next stage -- that is, half the nodes require no computation whatsoever!

**4.3 Convergence of Knot Insertion Algorithms.** Knot insertion is a corner cutting procedure. Therefore, since B-spline curves lie in the convex hull of their control points, if more and more knots are inserted, the control polygon will get closer and closer to the original B-spline curve. We shall now show that as the knot spacing approaches zero, the control polygons generated by any knot insertion procedure necessarily converge to the B-spline curve for the original control points. We shall then apply this result to construct new algorithms for rendering and intersecting B-spline curves and surfaces directly from knot insertion without converting to piecewise Bezier form. We will also apply this result to provide a straightforward proof of the variation diminishing property for B-spline curves.

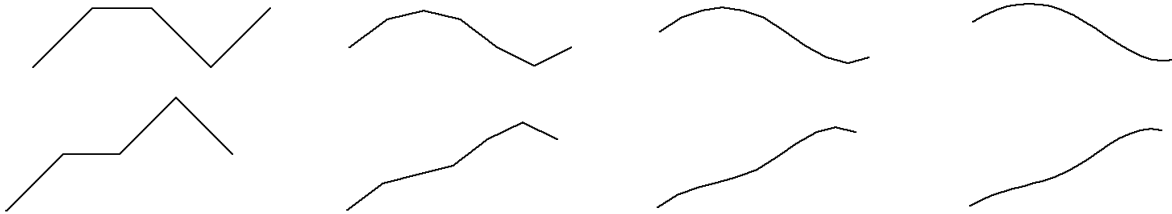
**Theorem 4.1:** *The control polygons generated by knot insertion converge to the B-spline curve for the original control polygon as the knot spacing approaches zero.*

Proof: Let  $S(t)$  be a B-spline curve of degree  $n$  with knots  $t_1, \dots, t_m$ , and let  $P_k(t)$  denote the

polynomial segment of the spline  $S(t)$  over the interval  $[t_k, t_{k+1}]$ . By the dual functional property, over the interval  $[t_k, t_{k+1}]$  the B-spline control points for the spline  $S(t)$  are given by the blossom values  $p_k(t_{k-n+1}, \dots, t_k), \dots, p_k(t_{k+1}, \dots, t_{k+n})$ . Suppose that knots are inserted so that the distance between any two consecutive knots gets arbitrarily small. Let  $t_1^n, \dots, t_d^n$  denote the new knot sequence after  $n$  iterations of knot insertion, and let  $[t_j^n, t_{j+1}^n]$  be a sequence of subintervals of  $[t_k, t_{k+1}]$  converging to a point  $\tau$  in the interval  $[t_k, t_{k+1}]$ . Again by the dual functional property, over the interval  $[t_j^n, t_{j+1}^n]$  the new B-spline control points are given by the blossom values  $p_k(t_{j-n+1}^n, \dots, t_j^n), \dots, p_k(t_{j+1}^n, \dots, t_{j+n}^n)$ . But  $p_k$  is a polynomial, so  $p_k$  is certainly a continuous function. Therefore as the distance between consecutive knots approaches zero,  $t_{j-n+1}^n, \dots, t_{j+n}^n \rightarrow \tau$ , so  $p_k(t_{j-n+1}^n, \dots, t_{j+n}^n) \rightarrow p_k(\tau, \dots, \tau) = P_k(\tau)$ . Therefore the control polygons generated by knot insertion converge to the B-spline curve for the original control polygon as the knot spacing approaches zero.



There is an analogue of Theorem 4.1 for tensor product B-spline surfaces. The proof is much the same, so we shall not repeat this proof here. In Figure 13, we illustrate the convergence of the control polygon under the Lane-Riesenfeld knot insertion algorithm for cubic B-spline curves.



**Figure 13:** Control polygons converging to B-spline curves. Here we illustrate the original control polygons (left) and the first three iterations of the Lane-Riesenfeld knot insertion algorithm for cubic B-spline curves with two polynomial segments (right).

We can now provide a straightforward proof of the variation diminishing property for B-spline curves without converting to piecewise Bezier form.

**Theorem 4.2:** *B-spline curves satisfy the variation diminishing property.*

Proof: Since knot insertion is a corner cutting procedure, the limit of knot insertion must be variation diminishing with respect to the original control polygon. But by Theorem 4.1, the B-

spline curve is the limit curve generated by iterating knot insertion, so B-spline curves are variation diminishing.



#### **4.4 Algorithms for Rendering and Intersecting B-Spline Curves and Surfaces Revisited.**

We do not need to convert from B-spline to piecewise Bezier form to generate algorithms for rendering and intersecting B-spline curves and surfaces. Instead we can apply global knot insertion procedures and rely on the convergence of the control polygon or polyhedron to the corresponding B-spline curve or surface as the knot spacing approaches zero. We begin by illustrating this approach for B-spline curves and then extend these techniques to tensor product B-spline surfaces.

##### *Rendering Algorithm -- B-Spline Curves*

Apply a global knot insertion procedure to generate a control polygon that approximates the B-spline curve to within the given tolerance.  
Render the control polygon.

##### *Intersection Algorithm -- B-Spline Curves*

If the convex hulls of the control points of two B-spline curves fail to intersect, then the curves themselves do not intersect.  
Otherwise if each B-spline curve can be approximated by the straight line segments joining the first and last points of each polynomial segment, then intersect these line segments.  
Otherwise apply a global knot insertion procedure to insert one new knot midway between each consecutive pair of the original knots and intersect the B-spline curves recursively.

Finding and intersecting two convex hulls of two sets of control points can be quite difficult and time consuming. In practice, the convex hulls in the intersection algorithm are typically replaced by bounding boxes which are much easier to compute and intersect than the actual convex hulls. Since the knot insertion algorithm converges rapidly, not much time is lost in the intersection algorithm by replacing convex hulls with bounding boxes.

To determine whether a B-spline curve can be approximated to within tolerance either by its control polygon or by the straight line segments joining the first and last points of each polynomial segment, it is sufficient, by the convex hull property, to test for each polynomial segment whether the corresponding control points all lie within tolerance of the line segment joining the first and last points of the segment. We can compute the end points of these line segments using the de Boor algorithm. We can then use the formula for the distance between a point  $P$  and a line  $L$  (see Lecture 11, Section 4.1.2) to test whether the control points all lie within some given tolerance of the line determined by the first and last points of the polynomial segment. In practice, after just a few

iterations of a global knot insertion procedure most if not all of the polynomial segments will be well approximated by their control polygons (see Figure 13). If there are still a few anomalous segments, we can switch from a global to a local knot insertion procedure to get a better approximation for these remaining segments of the B-spline curve.

We can also render or ray trace tensor product B-spline surfaces using global knot insertion. We begin with a triangulation algorithm.

#### *Triangulation Algorithm -- B-Spline Surfaces*

If each polynomial patch of the tensor product B-spline surface can be approximated to within tolerance by two triangles each determined by three of its four corner points and if the four boundary curves of the patch can be approximated by straight line segments joining the four corner points of the patch, then triangulate the B-spline patch by the triangles determined by these corner points of each of the polynomial patches.

Otherwise apply a global knot insertion procedure to insert one new knot midway between each consecutive pair of the original knots in both  $s$  and  $t$  and triangulate the surface recursively.

#### *Rendering Algorithm -- B-Spline Surfaces*

Triangulate the B-spline surface.

Apply your favorite shading and hidden surface algorithms to render the triangulated surface.

#### *Ray Tracing Algorithm -- B-Spline Surfaces*

If the ray does not intersect the convex hull of the control points of the B-spline surface, then the ray and the surface do not intersect.

Otherwise if the B-spline surface can be approximated to within tolerance by triangles each determined by three of the four corner points of each polynomial patch and if the four boundaries of each patch can be approximated by straight line segments joining the four corner points, then intersect the ray with these triangles.

Otherwise apply a global knot insertion procedure to insert one new knot midway between each consecutive pair of the original knots in both  $s$  and  $t$  and ray trace the surface recursively.

Keep the intersection closest to the eye.

Again finding the convex hull of the control points of a B-spline surface can be quite difficult and time consuming. In practice for surfaces, as with curves, the convex hull is replaced by a bounding box. Since the knot insertion algorithm converges rapidly, not much time is lost in the ray tracing algorithm by replacing convex hulls with bounding boxes.

To determine whether a polynomial patch on a tensor product B-spline surface can be approximated to within tolerance by a pair of triangles determined by the four corner points, it is sufficient, by the convex hull property, to test whether each control point lies within some tolerance of at least one of the two triangles. We can compute the corner points of these polynomial patches using the de Boor algorithm. We can then use the formula for the distance between a point  $P$  and a plane  $S$  (see Lecture 11, Section 4.1.3) to test whether the control points all lie within some given tolerance of the plane determined by the corner points of the polynomial patch. In practice, as with B-spline curves, after just a few iterations of a global knot insertion procedure most, if not all, of the polynomial patches will be well approximated by two triangles determined by their four corner points. Again if there are still a few anomalous patches, we can switch from a global to a local knot insertion procedure to get a better approximation for these remaining patches of the B-spline surface.

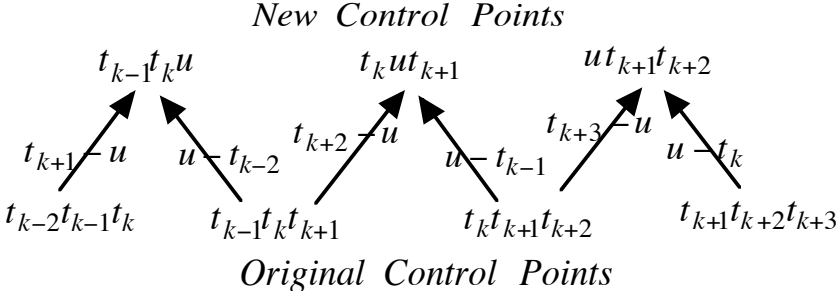
**5. Summary**

Knot insertion is a powerful technique for analyzing B-spline curves and surfaces. Techniques for converting to piecewise Bezier form, algorithms for rendering and intersecting B-spline curves and surfaces, and a proof of the variation diminishing property for B-spline curves can all be derived from knot insertion.

There are two types of knot insertion algorithms: local and global. Boehm’s algorithm and the Oslo algorithm are local procedures; the Lane-Riesenfeld algorithm and Schaefer’s algorithm are global procedures. Below we summarize the main properties of each of these procedures.

*Boehm’s Algorithm*

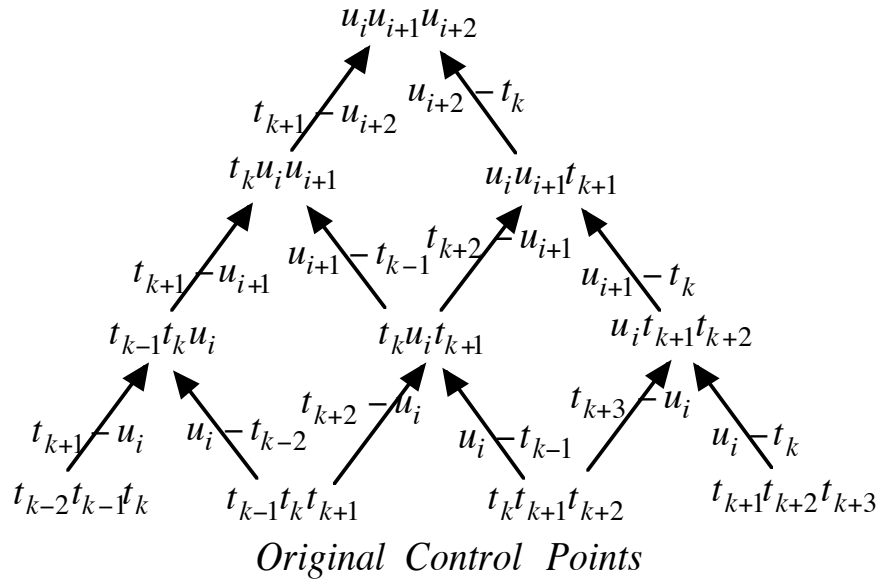
- Local knot insertion
- Inserts one new knot at a time
- Calculation equivalent to one level of the de Boor algorithm



**Figure 14:** Boehm’s algorithm for inserting a new knot  $u$  in the interval  $[t_k, t_{k+1}]$  into a cubic B-spline curve. The original control points are placed at the base of the diagram and the new control points emerge at the top of the diagram.

### Oslo Algorithm

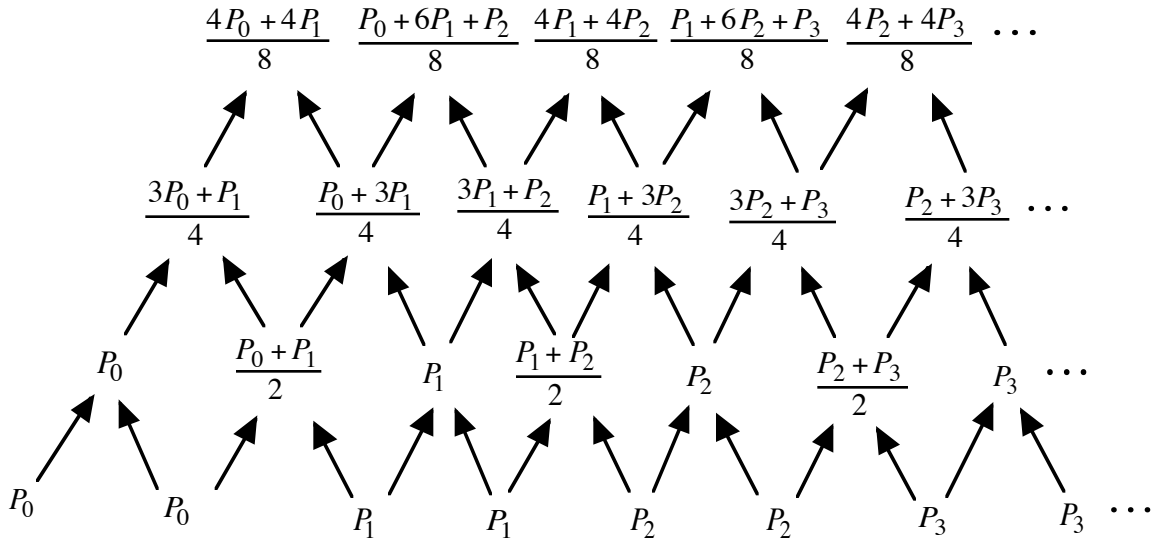
- Local knot insertion
- Computes one new control point at a time
- Calculation equivalent to the blossomed version of the de Boor algorithm



**Figure 15:** The Oslo algorithm for computing a new control point  $u_i u_{i+1} u_{i+2}$  in the interval  $[t_k, t_{k+1}]$  for a cubic B-spline curve. The original control points are placed at the base of the diagram and the new control point emerges at the apex of the diagram.

### Lane-Riesenfeld Algorithm

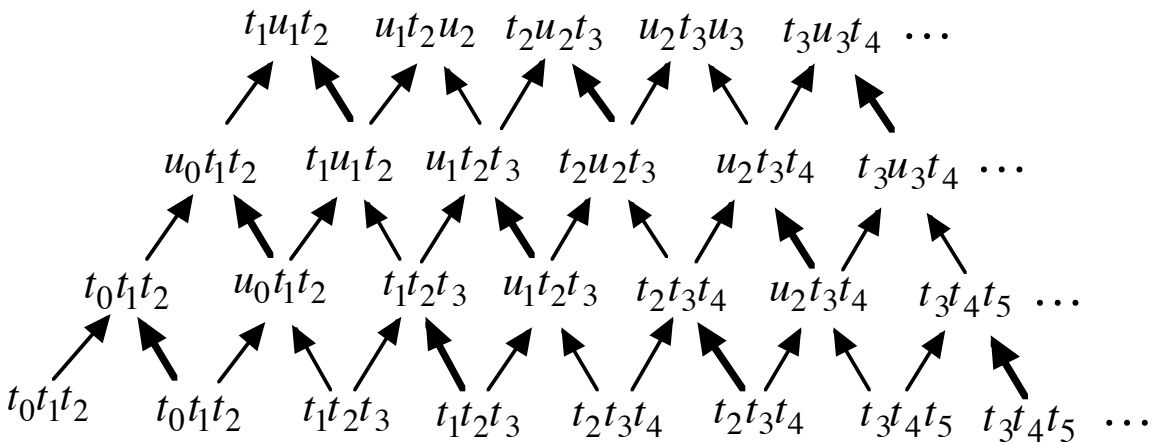
- Global knot insertion
- Works only for uniformly spaced knots
- Inserts one new knot midway between each consecutive pair of the original knots
- Two stage algorithm: split and average



**Figure 16:** The Lane-Riesenfeld algorithm for cubic B-spline curves with uniform knots. The original control points are split and placed at the base of the diagram. Three rounds of midpoint averaging are performed and the new control points then emerge at the top of the diagram.

*Schaefer's Algorithm*

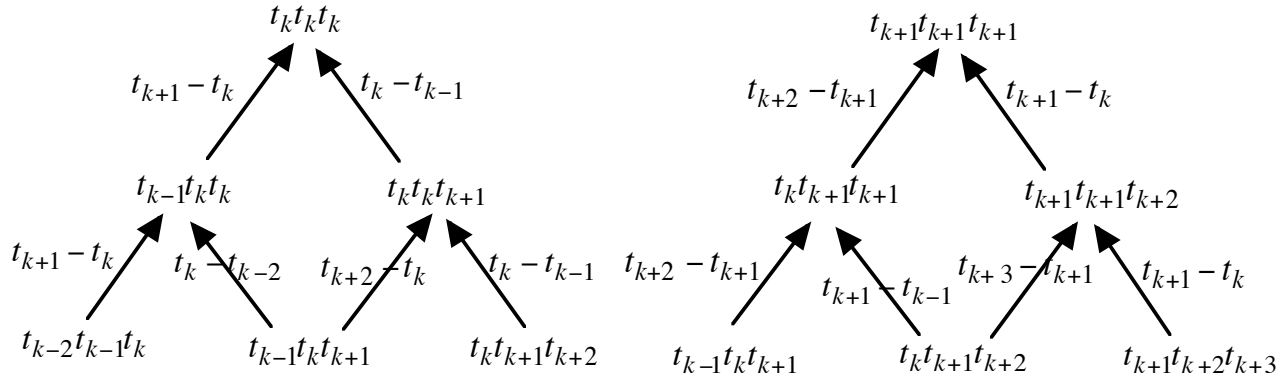
- Global knot insertion
- Works for arbitrary knots
- Inserts one new knot between each consecutive pair of the original knots
- Two stage algorithm: split and weighted average



**Figure 17:** Schaefer's knot insertion algorithm for cubic B-spline curves with arbitrary knots. The original control points are split and placed at the base of the diagram. Three rounds of weighted averaging are performed and the new control points emerge at the top of the diagram.

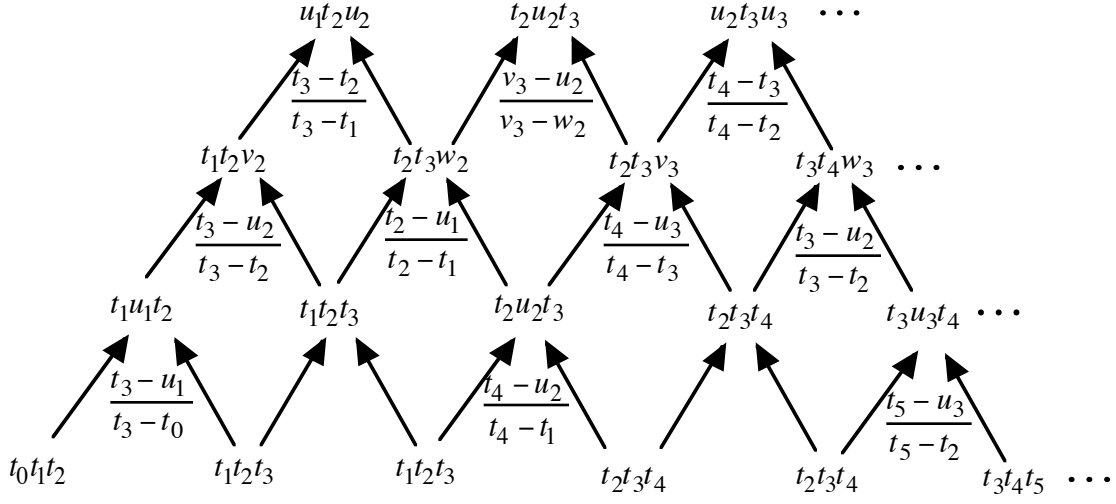
**Exercises:**

1. Show that the procedure diagrammed in Figure 18 can be applied to convert from cubic B-spline to piecewise Bezier form. Explain how this algorithm differs from the algorithm in Figure 5.



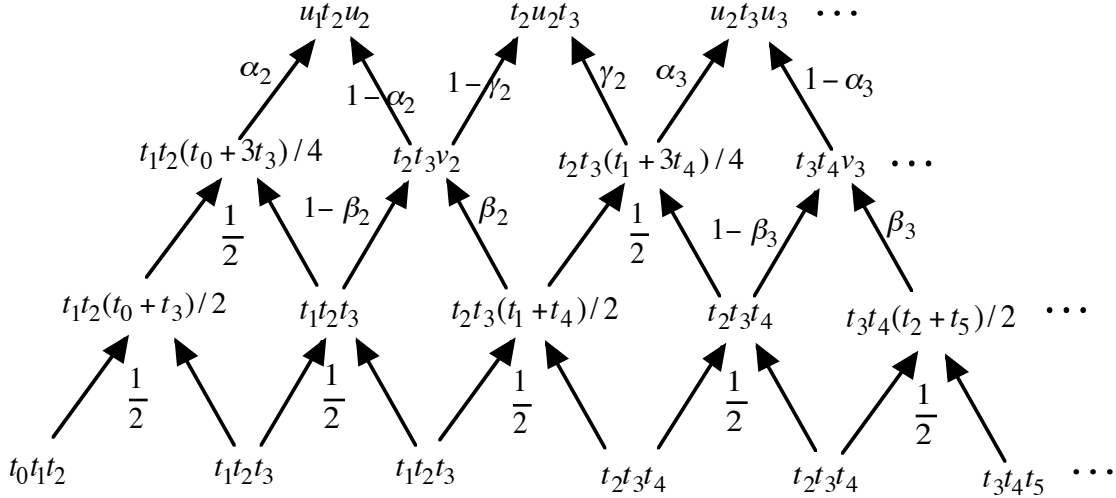
**Figure 18:** An algorithm to convert from cubic B-spline to piecewise cubic Bezier form.

2. Show that the Lane-Riesenfeld algorithm for quadratic B-splines with uniform knots has the following three step interpretation:
  - i. Split the control points.
  - ii. Convert to piecewise Bezier form.
  - iii Apply Boehm's knot insertion algorithm to insert the new knots.
3. Based on Exercise 2:
  - a. Develop an analogue of the Lane-Riesenfeld algorithm for quadratic B-splines with arbitrary knots.
  - b. Extend this algorithm to cubic and then to arbitrary degree B-splines with arbitrary knots.
4. Draw a blossoming diagram of Schaefer's knot insertion algorithm for quartic B-spline curves.
5. Consider the algorithm diagrammed in Figure 19. Show that:
  - a. this algorithm correctly inserts one new knot  $u_i$  between each consecutive pair  $t_i, t_{i+1}$  of the original knots in a cubic B-spline curve.
  - b. this algorithm reduces to the Lane-Riesenfeld algorithm if the knots are evenly spaced.
6. Consider the algorithm diagrammed in Figure 20. Show that:
  - a. this algorithm correctly inserts one new knot  $u_i$  between each consecutive pair  $t_i, t_{i+1}$  of the original knots in a cubic B-spline curve.
  - b. this algorithm reduces to the Lane-Riesenfeld algorithm if the knots are evenly spaced.



**Figure 19:** A global algorithm to insert knots into a cubic B-spline curve. The original knots are  $t_0, \dots, t_m$  and one new knot  $u_i$  is inserted between each consecutive pair  $t_i, t_{i+1}$  of the original knots. The expression inside each small triangle is the label on the left edge of the accompanying triangle; the label on the right edge is one minus the label on the left edge. Here

$$v_k = t_{k+1} - \frac{(t_{k+1} - u_{k-1})(t_{k+1} - u_k)}{(t_{k+1} - t_k)} \quad \text{and} \quad w_k = u_k - \frac{(u_k - t_{k-1})(t_k - u_{k-1})}{(t_k - t_{k-1})}.$$



**Figure 20:** Another global algorithm to insert knots into a cubic B-spline curve. The original knots are  $t_0, \dots, t_m$  and one new knot  $u_i$  is inserted between each consecutive pair  $t_i, t_{i+1}$  of the original knots. Here  $\alpha_k = \frac{4(t_{k+1} - u_{k-1})(t_{k+1} - u_k)}{(t_{k+1} - t_{k-2})(t_{k+1} - t_{k-1})}$ ,  $\beta_k = \frac{2(u_k - t_{k-1})(u_{k-1} - t_{k-1})}{(t_{k+2} - t_{k-1})(t_{k+1} - t_{k-1})(1 - \alpha_k)}$

$$\gamma_k = \frac{4(u_k - v_k)}{t_{k-1} + 3t_{k+2} - 4v_k}, \quad \text{and} \quad v_k = t_{k-1} + \frac{\beta_k(t_{k+2} - t_{k-1})}{2}.$$

## **Programming Project:**

### 1. *B-Spline Curves and Surfaces*

Implement a modeling system based on B-spline curves and surfaces in your favorite programming language using your favorite API.

- a. Include algorithms for rendering B-spline curves and surfaces based on
  - i. converting to piecewise Bezier form
  - ii. global knot insertion algorithms.
- b. Incorporate the ability to move control points interactively and have the B-spline curve or surface adjust in real time.
- c. Create a new font using B-spline curves.
- d. Build some interesting freeform shapes using B-spline surfaces.