Curves and Surfaces for CAGD

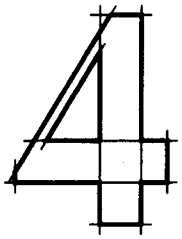
A Practical Guide

Fifth Edition

Gerald Farin
Arizona State University

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ACADEMIC PRESS
A Harcourt Science and Technology Company
SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

The de Casteljau Algorithm



The algorithm described in this chapter is probably the most fundamental one in the field of curve and surface design, yet it is surprisingly simple. Its main attraction is the beautiful interplay between geometry and algebra: a very intuitive geometric construction leads to a powerful theory.

Historically, it is with this algorithm that the work of de Casteljau started in 1959. The only written evidence is in [145] and [146], both technical reports that are not easily accessible. De Casteljau's work went unnoticed until W. Boehm obtained copies of the reports in 1975. Since then, de Casteljau's work has gained more popularity.

4.1 Parabolas

We give a simple construction for the generation of a parabola; the straightforward generalization will then lead to Bézier curves. Let $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$ be any three points in $\mathbb{E}^3$, and let $t \in \mathbb{R}$. Construct

$$\mathbf{b}_0^1(t) = (1-t)\mathbf{b}_0 + t\mathbf{b}_1,$$

$$\mathbf{b}_1^1(t) = (1-t)\mathbf{b}_1 + t\mathbf{b}_2,$$

$$\mathbf{b}_0^2(t) = (1-t)\mathbf{b}_0^1(t) + t\mathbf{b}_1^1(t).$$

Inserting the first two equations into the third one, we obtain

$$\mathbf{b}_0^2(t) = (1-t)^2\mathbf{b}_0 + 2t(1-t)\mathbf{b}_1 + t^2\mathbf{b}_2. \tag{4.1}$$

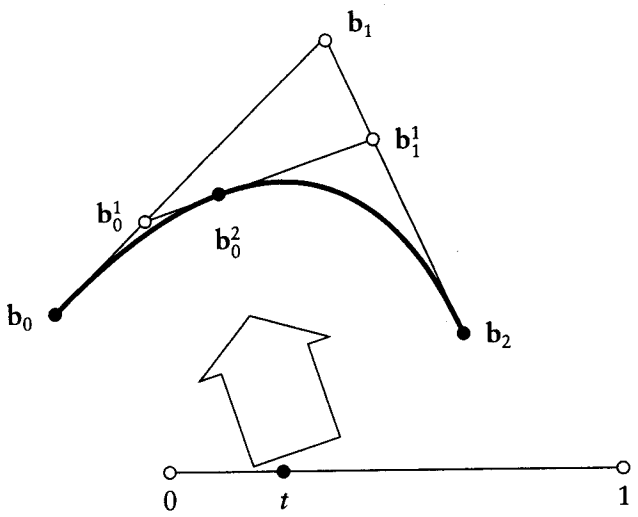


Figure 4.1 Parabolas: construction by repeated linear interpolation.

This is a quadratic expression in t (the superscript denotes the degree), and so $\mathbf{b}_0^2(t)$ traces out a *parabola* as t varies from $-\infty$ to $+\infty$. We denote this parabola by $\mathbf{b}^2$. This construction consists of *repeated linear interpolation*; its geometry is illustrated in Figure 4.1. For t between 0 and 1, $\mathbf{b}^2(t)$ is inside the triangle formed by $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$; in particular, $\mathbf{b}^2(0) = \mathbf{b}_0$ and $\mathbf{b}^2(1) = \mathbf{b}_2$.

Inspecting the ratios of points in Figure 4.1, we see that

$$\text{ratio}(\mathbf{b}_0, \mathbf{b}_0^1, \mathbf{b}_1) = \text{ratio}(\mathbf{b}_1, \mathbf{b}_1^1, \mathbf{b}_2) = \text{ratio}(\mathbf{b}_0^1, \mathbf{b}_0^2, \mathbf{b}_1^1) = t/(1-t).$$

Thus our construction of a parabola is *affinely invariant* because piecewise linear interpolation is affinely invariant; see Section 3.2.

We also note that a parabola is a plane curve, since $\mathbf{b}^2(t)$ is always a barycentric combination of three points, as is clear from inspecting (4.1). A parabola is a special case of *conic sections*, which will be discussed in Chapter 12.

Finally we state a theorem from analytic geometry, closely related to our parabola construction. Let $\mathbf{a}, \mathbf{b}, \mathbf{c}$ be three distinct points on a parabola. Let the tangent at $\mathbf{b}$ intersect the tangents at $\mathbf{a}$ and $\mathbf{c}$ in $\mathbf{e}$ and $\mathbf{f}$, respectively. Let the tangents at $\mathbf{a}$ and $\mathbf{c}$ intersect in $\mathbf{d}$. Then $\text{ratio}(\mathbf{a}, \mathbf{e}, \mathbf{d}) = \text{ratio}(\mathbf{e}, \mathbf{b}, \mathbf{f}) = \text{ratio}(\mathbf{d}, \mathbf{f}, \mathbf{c})$. This *three tangent theorem* describes a property of parabolas; the de Casteljau algorithm can be viewed as the constructive counterpart. Figure 4.1, although using a different notation, may serve as an illustration of the theorem.

4.2 The de Casteljau Algorithm

Parabolas are plane curves. However, many applications require true space curves.[1] For those purposes, the previous construction for a parabola can be generalized to generate a polynomial curve of arbitrary degree n:

de Casteljau algorithm:

Given: $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{E}^3$ and $t \in \mathbb{R}$,

set

$$\mathbf{b}_i^r(t) = (1-t)\mathbf{b}_i^{r-1}(t) + t\mathbf{b}_{i+1}^{r-1}(t) \quad \begin{cases} r = 1, \ldots, n \\ i = 0, \ldots, n-r \end{cases} \tag{4.2}$$

and $\mathbf{b}_i^0(t) = \mathbf{b}_i$. Then $\mathbf{b}_0^n(t)$ is the point with parameter value t on the *Bézier curve* $\mathbf{b}^n$, hence $\mathbf{b}^n(t) = \mathbf{b}_0^n(t)$.

The polygon P formed by $\mathbf{b}_0, \ldots, \mathbf{b}_n$ is called the *Bézier polygon* or *control polygon* of the curve $\mathbf{b}^n$.[2] Similarly, the polygon vertices $\mathbf{b}_i$ are called *control points* or *Bézier points*. Figure 4.2 illustrates the cubic case.

Sometimes we also write $\mathbf{b}^n(t) = \mathbf{B}[\mathbf{b}_0, \ldots, \mathbf{b}_n; t] = \mathbf{B}[\mathbf{P}; t]$ or, shorter, $\mathbf{b}^n = \mathbf{B}[\mathbf{b}_0, \ldots, \mathbf{b}_n] = \mathbf{BP}$. This notation[3] defines B to be the (linear) operator that associates the Bézier curve with its control polygon. We say that the curve $\mathbf{B}[\mathbf{b}_0, \ldots, \mathbf{b}_n]$ is the *Bernstein–Bézier approximation* to the control polygon, a terminology borrowed from approximation theory; see also Section 6.9.

The intermediate coefficients $\mathbf{b}_i^r(t)$ are conveniently written into a triangular array of points, the *de Casteljau scheme*. We give the example of the cubic case:

$$\begin{array}{cccc} \mathbf{b}_0 & & & \\ \mathbf{b}_1 & \mathbf{b}_0^1 & & \\ \mathbf{b}_2 & \mathbf{b}_1^1 & \mathbf{b}_0^2 & \\ \mathbf{b}_3 & \mathbf{b}_2^1 & \mathbf{b}_1^2 & \mathbf{b}_0^3. \end{array} \tag{4.3}$$

This triangular array of points seems to suggest the use of a two-dimensional array in writing code for the de Casteljau algorithm. That would be a waste of

1 Compare the comments by P. Bézier in Chapter 1!

2 In the cubic case, there are four control points; they form a tetrahedron in the 3D case. This tetrahedron was already mentioned by W. Blaschke [65] in 1923; he called it "osculating tetrahedron."

3 This notation should not be confused with the blossoming notation used later.

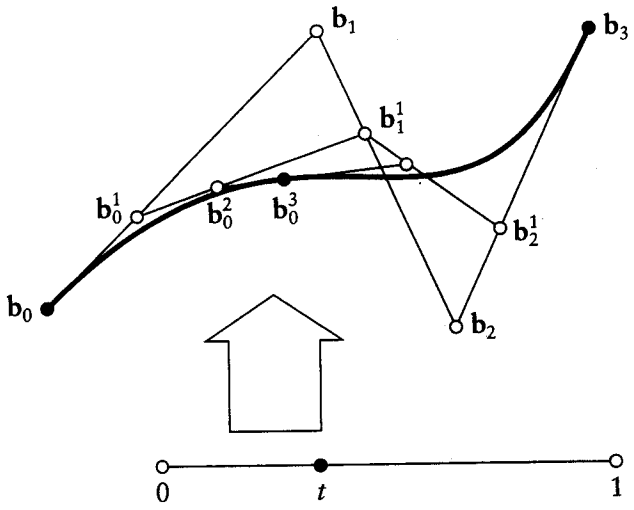


Figure 4.2 The de Casteljau algorithm: the point $\mathbf{b}_0^3(t)$ is obtained from repeated linear interpolation. The cubic case $n = 3$ is shown for $t = 1/3$.

Example 4.1 **Computing a point on a Bézier curve with the de Casteljau algorithm.**

A de Casteljau scheme for a planar cubic and for $t = \frac{1}{2}$:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 \\ 2 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} 8 \\ 2 \end{bmatrix} \quad \begin{bmatrix} 4 \\ 2 \end{bmatrix} \quad \begin{bmatrix} 2 \\ \frac{3}{2} \end{bmatrix}$$
$$\begin{bmatrix} 4 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 6 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 5 \\ \frac{3}{2} \end{bmatrix} \quad \begin{bmatrix} \frac{7}{2} \\ \frac{3}{2} \end{bmatrix}$$

storage, however: it is sufficient to use the left column only and to overwrite it appropriately.

For a numerical example, see Example 4.1. Figure 4.3 shows 60 evaluations of a Bézier curve. The intermediate points $\mathbf{b}_i^r$ are also plotted and connected.[4]

4 Although the control polygon of the figure is symmetric, the plot is not. This is due to the organization of the plotting algorithm.

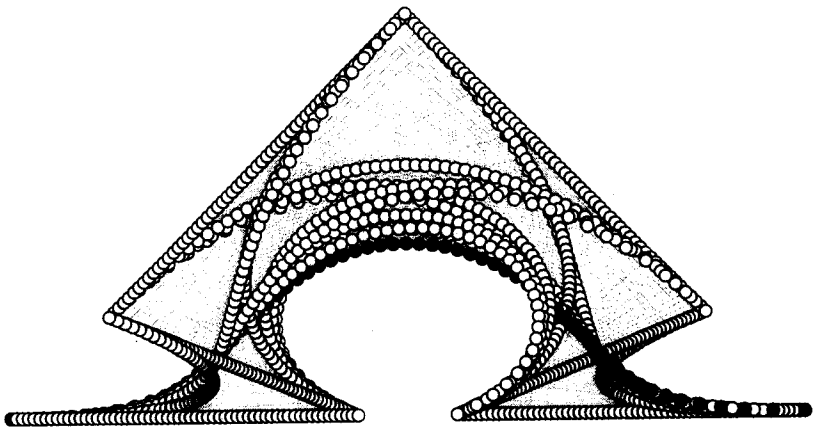


Figure 4.3 The de Casteljau algorithm: 60 points are computed on a degree six curve; all intermedediate points $\mathbf{b}_i^r$ are shown.

4.3 Some Properties of Bézier Curves

The de Casteljau algorithm allows us to infer several important properties of Bézier curves. We will infer these properties from the geometry underlying the algorithm. In the next chapter, we will show how they can also be derived analytically.

Affine invariance. Affine maps were discussed in Section 2.2. They are in the tool kit of every CAD system: objects must be repositioned, scaled, and so on. An important property of Bézier curves is that they are invariant under affine maps, which means that the following two procedures yield the same result: (1) first, compute the point $\mathbf{b}^n(t)$ and then apply an affine map to it; (2) first, apply an affine map to the control polygon and then evaluate the mapped polygon at parameter value t.

Affine invariance is, of course, a direct consequence of the de Casteljau algorithm: the algorithm is composed of a sequence of linear interpolations (or, equivalently, of a sequence of affine maps). These are themselves affinely invariant, and so is a finite sequence of them.

Let us discuss a practical aspect of affine invariance. Suppose we plot a cubic curve $\mathbf{b}^3$ by evaluating at 100 points and then plotting the resulting point array. Suppose now that we would like to plot the curve after a rotation has been applied to it. We can take the 100 computed points, apply the rotation to each of them, and plot. Or, we can apply the rotation to the 4 control points, then evaluate 100 times and plot. The first method needs 100 applications of the rotation, whereas the second needs only 4!

Affine invariance may not seem to be a very exceptional property for a useful curve scheme; in fact, it is not straightforward to think of a curve scheme that does not have it (exercise!). It is perhaps worth noting that Bézier curves do *not* enjoy another, also very important, property: they are not *projectively invariant*. Projective maps are used in computer graphics when an object is to be rendered realistically. So if we try to make life easy and simplify a perspective map of a Bézier curve by mapping the control polygon and then computing the curve, we have actually cheated: that curve is not the perspective image of the original curve! More details on perspective maps can be found in Chapter 12.

Invariance under affine parameter transformations. Very often, one thinks of a Bézier curve as being defined over the interval $[0, 1]$. This is done because it is convenient, not because it is necessary: the de Casteljau algorithm is "blind" to the actual interval that the curve is defined over because it uses ratios only. One may therefore think of the curve as being defined over any arbitrary interval $a \le u \le b$ of the real line—after the introduction of local coordinates $t = (u - a)/(b - a)$, the algorithm proceeds as usual. This property is inherited from the linear interpolation process (3.9). The corresponding generalized de Casteljau algorithm is of the form:

$$\mathbf{b}_i^r(u) = \frac{b-u}{b-a}\mathbf{b}_i^{r-1}(u) + \frac{u-a}{b-a}\mathbf{b}_{i+1}^{r-1}(u). \tag{4.4}$$

The transition from the interval $[0, 1]$ to the interval $[a, b]$ is an *affine map*. Therefore, we can say that Bézier curves are invariant under affine parameter transformations. Sometimes, one sees the term *linear parameter transformation* in this context, but this terminology is not quite correct: the transformation of the interval $[0, 1]$ to $[a, b]$ typically includes a translation, which is not a linear map.

Convex hull property. For $t \in [0, 1]$, $\mathbf{b}^n(t)$ lies in the convex hull (see Figure 2.3) of the control polygon. This follows since every intermediate $\mathbf{b}_i^r$ is obtained as a convex barycentric combination of previous $\mathbf{b}_j^{r-1}$—at no step of the de Casteljau algorithm do we produce points outside the convex hull of the $\mathbf{b}_i$.

A simple consequence of the convex hull property is that a planar control polygon always generates a planar curve.

The importance of the convex hull property lies in what is known as *interference checking*. Suppose we want to know if two Bézier curves intersect each other—for example, each might represent the path of a robot arm, and our aim is to make sure that the two paths do not intersect, thus avoiding expensive collisions of the robots. Instead of actually computing a possible intersection, we can perform a much cheaper test: circumscribe the smallest possible box around the control polygon of each curve such that it has its edges parallel to some coordinate system. Such boxes are called *minmax boxes*, since their faces are created by the minimal and maximal coordinates of the control polygons. Clearly each box contains its control polygon, and, by the convex hull property, also the corresponding Bézier curve. If we can verify that the two boxes do not overlap (a trivial test), we are assured that the two curves do not intersect. If the boxes do overlap, we would have to perform more checks on the curves. The possibility for a quick decision of no interference is extremely important, since in practice one often has to check one object against thousands of others, most of which can be labeled "no interference" by the minmax box test.[5]

Endpoint interpolation. The Bézier curve passes through $\mathbf{b}_0$ and $\mathbf{b}_n$: we have $\mathbf{b}^n(0) = \mathbf{b}_0, \mathbf{b}^n(1) = \mathbf{b}_n$. This is easily verified by writing down the scheme (4.3) for the cases $t = 0$ and $t = 1$. In a design situation, the endpoints of a curve are certainly two very important points. It is therefore essential to have direct control over them, which is assured by endpoint interpolation.

Designing with Bézier curves. Figure 4.4 shows two Bézier curves. From the inspection of these examples, one gets the impression that in some sense the Bézier curve "mimics" the Bézier polygon—this statement will be made more precise later. It is the reason Bézier curves provide such a handy tool for the *design* of curves: to reproduce the shape of a hand-drawn curve, it is sufficient to specify a control polygon that somehow "exaggerates" the shape of the curve. One lets the computer draw the Bézier curve defined by the polygon, and, if necessary, adjusts the location (possibly also the number) of the polygon vertices. Typically, an experienced person will reproduce a given curve after two to three iterations of this *interactive* procedure.

5 It is possible to create volumes (or areas, in the 2D case) that hug the given curve closer than the minmax box does. See Sederberg et al. [560].

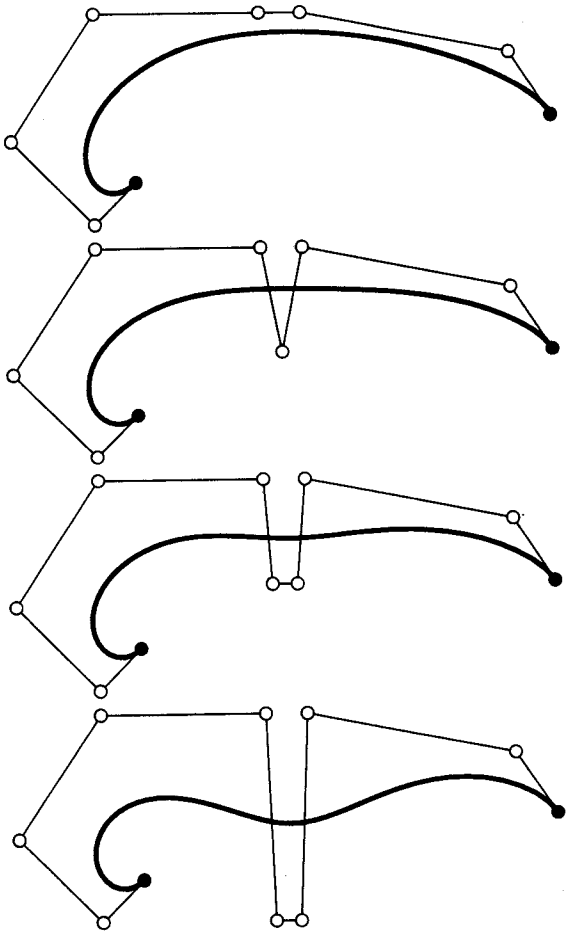


Figure 4.4 Bézier curves: some examples.

4.4 The Blossom

In recent years, a new way to look at Bézier curves has been developed; it is called the principle of *blossoming*. This principle was independently developed by de Casteljau [147] and Ramshaw [498], [499]. Other literature includes Seidel [562], [565], [566]; DeRose and Goldman [165]; Boehm [75]; Lee [379]; and Gallier [252].

Blossoms were introduced in Section 3.4. They are closely related to the de Casteljau algorithm: in column r, do not again perform a de Casteljau step for parameter value t, but use a new value t_r. Restricting ourselves to the cubic case, we obtain:

$$\begin{array}{llll} \mathbf{b}_0 & & & \\ \mathbf{b}_1 & \mathbf{b}_0^1[t_1] & & \\ \mathbf{b}_2 & \mathbf{b}_1^1[t_1] & \mathbf{b}_0^2[t_1,t_2] & \\ \mathbf{b}_3 & \mathbf{b}_2^1[t_1] & \mathbf{b}_1^2[t_1,t_2] & \mathbf{b}_0^3[t_1,t_2,t_3]. \end{array} \tag{4.5}$$

The resulting point $\mathbf{b}_0^3[t_1, t_2, t_3]$ is now a function of three independent variables; thus it no longer traces out a curve, but a region of $\mathbb{E}^3$. This trivariate function $\mathbf{b}[\cdot, \cdot, \cdot]$ is called the blossom from Section 3.4. The original curve is recovered if we set all three arguments equal: $t = t_1 = t_2 = t_3$.

To understand the blossom better, we now evaluate it for several special arguments. We already know, of course, that $\mathbf{b}[0, 0, 0] = \mathbf{b}_0$ and $\mathbf{b}[1, 1, 1] = \mathbf{b}_3$. Let us start with $[t_1, t_2, t_3] = [0, 0, 1]$. The scheme (4.5) reduces to:

$$\begin{array}{llll} \mathbf{b}_0 & & & \\ \mathbf{b}_1 & \mathbf{b}_0 & & \\ \mathbf{b}_2 & \mathbf{b}_1 & \mathbf{b}_0 & \\ \mathbf{b}_3 & \mathbf{b}_2 & \mathbf{b}_1 & \mathbf{b}_1 = \mathbf{b}[0,0,1]. \end{array} \tag{4.6}$$

Similarly, we can show that $\mathbf{b}[0, 1, 1] = \mathbf{b}_2$. Thus the original Bézier points can be found by evaluating the curve's blossom at arguments consisting only of 0's and 1's.

But the remaining entries in (4.3) may also be written as values of the blossom for special arguments. For instance, setting $[t_1, t_2, t_3] = [0, 0, t]$, we have the scheme

$$\begin{array}{llll} \mathbf{b}_0 & & & \\ \mathbf{b}_1 & \mathbf{b}_0 & & \\ \mathbf{b}_2 & \mathbf{b}_1 & \mathbf{b}_0 & \\ \mathbf{b}_3 & \mathbf{b}_2 & \mathbf{b}_1 & \mathbf{b}_0^1 = \mathbf{b}[0,0,t]. \end{array} \tag{4.7}$$

Continuing in the same manner, we may write the complete scheme (4.3) as:

$$\begin{array}{llll} \mathbf{b}_0 = \mathbf{b}[0,0,0] & & & \\ \mathbf{b}_1 = \mathbf{b}[0,0,1] & \mathbf{b}[0,0,t] & & \\ \mathbf{b}_2 = \mathbf{b}[0,1,1] & \mathbf{b}[0,t,1] & \mathbf{b}[0,t,t] & \\ \mathbf{b}_3 = \mathbf{b}[1,1,1] & \mathbf{b}[t,1,1] & \mathbf{b}[t,t,1] & \mathbf{b}[t,t,t]. \end{array} \tag{4.8}$$

This is easily generalized to arbitrary degrees, where we can also express the Bézier points as blossom values:

$$\mathbf{b}_i = \mathbf{b}[0^{<n-i>}, 1^{<i>}], \tag{4.9}$$

where $t^{<r>}$ means that t appears r times as an argument. For example, $\mathbf{b}[0^{<1>}, t^{<2>}, 1^{<0>}] = \mathbf{b}[0, t, t]$.

The de Casteljau recursion (4.2) can now be expressed in terms of the blossom $\mathbf{b}$:

$$\mathbf{b}[0^{<n-r-i>}, t^{<r>}, 1^{<i>}] = (1-t)\mathbf{b}[0^{<n-r-i+1>}, t^{<r-1>}, 1^{<i>}] + t\mathbf{b}[0^{<n-r-i>}, t^{<r-1>}, 1^{<i+1>}]. \tag{4.10}$$

The point on the curve is given by $\mathbf{b}[t^{<n>}]$.

We may also consider the blossom of a Bézier curve that is not defined over $[0, 1]$ but over the more general interval $[a, b]$. Proceeding exactly as above—but now using (4.4)—we find that the Bézier points $\mathbf{b}_i$ are found as the blossom values

$$\mathbf{b}_i = \mathbf{b}[a^{<n-i>}, b^{<i>}]. \tag{4.11}$$

Thus a cubic over $u \in [a, b]$ has Bézier points $\mathbf{b}[a, a, a], \mathbf{b}[a, a, b], \mathbf{b}[a, b, b], \mathbf{b}[b, b, b]$. If the original Bézier curve was defined over $[0, 1]$, the Bézier points of the one corresponding to $[a, b]$ are simply found by four calls to a blossom routine! See also Figure 4.5.

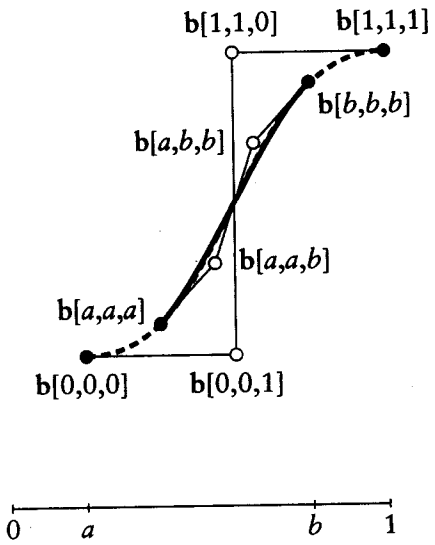


Figure 4.5 Subdivision: the relevant blossom values.

We may also find explicit formulas for blossoms; here is the case of a cubic:

$$
\begin{aligned}
&\mathbf{b}[t_1, t_2, t_3] \\
&= (1 - t_1)\mathbf{b}[0, t_2, t_3] + t_1\mathbf{b}[1, t_2, t_3] \\
&= (1 - t_1)\big[(1 - t_2)\mathbf{b}[0, 0, t_3] + t_2\mathbf{b}[0, 1, t_3]\big] + t_1\big[(1 - t_2)\mathbf{b}[0, 1, t_3] \\
&\quad + t_2\mathbf{b}[1, 1, t_3]\big] \\
&= \mathbf{b}[0, 0, 0](1 - t_1)(1 - t_2)(1 - t_3) \\
&\quad + \mathbf{b}[0, 0, 1]\big[(1 - t_1)(1 - t_2)t_3 + (1 - t_1)t_2(1 - t_3) + t_1(1 - t_2)(1 - t_3)\big] \\
&\quad + \mathbf{b}[0, 1, 1]\big[t_1t_2(1 - t_3) + t_1(1 - t_2)t_3 + (1 - t_1)t_2t_3\big] \\
&\quad + \mathbf{b}[1, 1, 1]t_1t_2t_3.
\end{aligned}
$$

For each step, we have exploited the fact that blossoms are multiaffine, following the inductive proof of the Leibniz equation (3.22).

We should add that every multivariate polynomial function may be interpreted as the blossom of a Bézier curve—as long as it is both symmetric and multiaffine.

4.5 Implementation

The header of the de Casteljau algorithm program is:

```
float decas(degree,coeff,t)
/*   uses  de Casteljau to compute one coordinate
     value of a  Bezier curve. Has to be called
     for each coordinate  (x,y, and/or z) of a control polygon.
Input:   degree:  degree of curve.
         coeff:   array with coefficients of curve.
         t:       parameter value.
Output: coordinate value.
*/
```

This procedure invites several comments. First, we see that it requires the use of an auxiliary array `coeffa`. Moreover, this auxiliary array has to be filled for each function call! So on top of the already high computational cost of the de Casteljau algorithm, we add another burden to the routine, keeping it from being very efficient. A faster evaluation method is given at the end of the next chapter.

To plot a Bézier curve, we would then call the routine several times:

```
void bez_to_points(degree,npoints,coeff,points)
/*     Converts Bezier curve into point sequence. Works on
       one coordinate only.
 Input:   degree:  degree of curve.
          npoints: # of coordinates to be generated. (counting
                   from 0!)
          coeff:   coordinates of control polygon.

Output:  points:   coordinates of points on curve.

     Remark: For a 2D curve, this routine needs to be called twice,
          once for the x-coordinates and once for y.
*/
```

The last subroutine has to be called once for each coordinate, that is, two or three times. The main program `decasmain.c` on the enclosed disk gives an example of how to use it and how to generate postscript output.

4.6 Problems

1 Suppose a planar Bézier curve has a control polygon that is symmetric with respect to the y-axis. Is the curve also symmetric with respect to the y-axis? Be sure to consider the control polygon $(-1, 0), (0, 1), (1, 1), (0, 2), (0, 1), (-1, 1), (0, 2), (0, 1), (1, 0)$. Generalize to other symmetry properties.

2 Use the de Casteljau algorithm to design a curve of degree four that has its middle control point on the curve. More specifically, try to achieve

$$\mathbf{b}_2 = \mathbf{b}_0^4\left(\frac{1}{2}\right).$$

Five collinear control points are a solution; try to be more ambitious!

*** 3** The de Casteljau algorithm may be formulated as

$$\mathbf{B}[\mathbf{b}_0, \ldots, \mathbf{b}_n; t] = (1-t)\mathbf{B}[\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}; t] + t\mathbf{B}[\mathbf{b}_1, \ldots, \mathbf{b}_n; t].$$

Show that the computation count is exponential (in terms of the degree) if you implement such a recursive algorithm in a language like C.

* **4** Show that every nonplanar cubic in $\mathbb{E}^3$ can be obtained as an affine map of the *standard cubic* (see Boehm [70])

$$\mathbf{x}(t) = \begin{bmatrix} t \\ t^2 \\ t^3 \end{bmatrix}.$$

P1 Write an experimental program that replaces $(1-t)$ and t in the recursion (4.2) by $[1-f(t)]$ and $f(t)$, where f is some "interesting" function. Change the routine decas accordingly and comment on your results.

P2 Rewrite the routine decas to handle blossoms. Evaluate and plot for some "interesting" arguments.

P3 Experiment with the data set `outline_2D.dat` on the floppy: try to recapture its shape using one, two, and four Bézier curves. These curves should have decreasing degrees as you use more of them.

P4 Then repeat the previous problem with `outline_3D.dat`. This data set is three dimensional, and you will have to use (at least) two views as you approximate the data points. The points, by the way, are taken from the outline of a high heel shoe sole.