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 Bernstein Form of a Bézier Curve

5

Bézier curves can be defined by a recursive algorithm, which is how de Casteljau first developed them. It is also necessary, however, to have an *explicit* representation for them; this will facilitate further theoretical development considerably.

5.1 Bernstein Polynomials

We will express Bézier curves in terms of *Bernstein polynomials*, defined explicitly by

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \tag{5.1}$$

where the binomial coefficients are given by

$$\binom{n}{i} = \begin{cases} \frac{n!}{i!(n-i)!} & \text{if} \quad 0 \le i \le n \\ 0 & \text{else.} \end{cases}$$

There is a fair amount of literature on these polynomials. We cite just a few: Bernstein [53], Lorentz [399], Davis [133], and Korovkin [364]. An extensive bibliography is given in Gonska and Meier [269].

Before we explore the importance of Bernstein polynomials to Bézier curves, let us first examine them more closely. One of their important properties is that they satisfy the following recursion:

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \tag{5.2}$$

with

$$B_0^0(t) \equiv 1 \tag{5.3}$$

and

$$B_j^n(t) \equiv 0 \quad \text{for} \quad j \notin \{0, \ldots, n\}. \tag{5.4}$$

The proof is simple:

$$\begin{aligned} B_i^n(t) &= \binom{n}{i} t^i (1-t)^{n-i} \\ &= \binom{n-1}{i} t^i (1-t)^{n-i} + \binom{n-1}{i-1} t^i (1-t)^{n-i} \\ &= (1-t) B_i^{n-1}(t) + t B_{i-1}^{n-1}(t). \end{aligned}$$

Another important property is that Bernstein polynomials form a *partition of unity*:

$$\sum_{j=0}^{n} B_j^n(t) \equiv 1. \tag{5.5}$$

This fact is proved with the help of the binomial theorem:

$$1 = [t + (1-t)]^n = \sum_{j=0}^{n} \binom{n}{j} t^j (1-t)^{n-j} = \sum_{j=0}^{n} B_j^n(t).$$

Figure 5.1 shows the family of the four cubic Bernstein polynomials. Note that the B_i^n are nonnegative over the interval $[0, 1]$.

We are now ready to see why Bernstein polynomials are important for the development of Bézier curves. Recall that a Bézier curve may be written as $\mathbf{b}[t^{<n>}]$ in blossom form. Since $t = (1-t) \cdot 0 + t \cdot 1$, the blossom may be expressed as $\mathbf{b}[(1-t) \cdot 0 + t \cdot 1)^{<n>}]$, and now the Leibniz formula (3.22) directly yields

$$\mathbf{b}(t) = \mathbf{b}[t^{<n>}] = \sum_{i=0}^{n} \mathbf{b}_i B_i^n(t) \tag{5.6}$$

since $\mathbf{b}_i = \mathbf{b}[0^{<n-i>}, 1^{<i>}]$ according to (4.9).

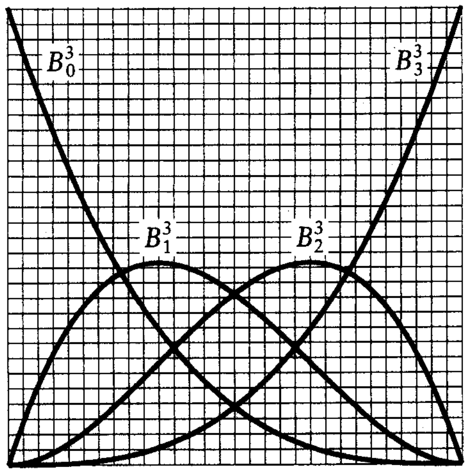

Figure 5.1 Bernstein polynomials: the cubic case.

Similarly, the intermediate de Casteljau points $\mathbf{b}_i^r$ can be expressed in terms of Bernstein polynomials of degree r:

$$\mathbf{b}_i^r(t) = \sum_{j=0}^{r} \mathbf{b}_{i+j} B_j^r(t). \tag{5.7}$$

This follows directly from

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

and the Leibniz formula.

Equation (5.7) shows exactly how the intermediate point $\mathbf{b}_i^r$ depends on the given Bézier points $\mathbf{b}_i$. Figure 5.2 shows how these intermediate points form Bézier curves themselves.

With the intermediate points $\mathbf{b}_i^r$ at hand, we can write a Bézier curve in the form

$$\mathbf{b}^n(t) = \sum_{i=0}^{n-r} \mathbf{b}_i^r(t) B_i^{n-r}(t). \tag{5.8}$$

This is to be interpreted as follows: first, compute r levels of the de Casteljau algorithm with respect to t. Then, interpret the resulting points $\mathbf{b}_i^r(t)$ as control points of a Bézier curve of degree $n - r$ and evaluate it at t.

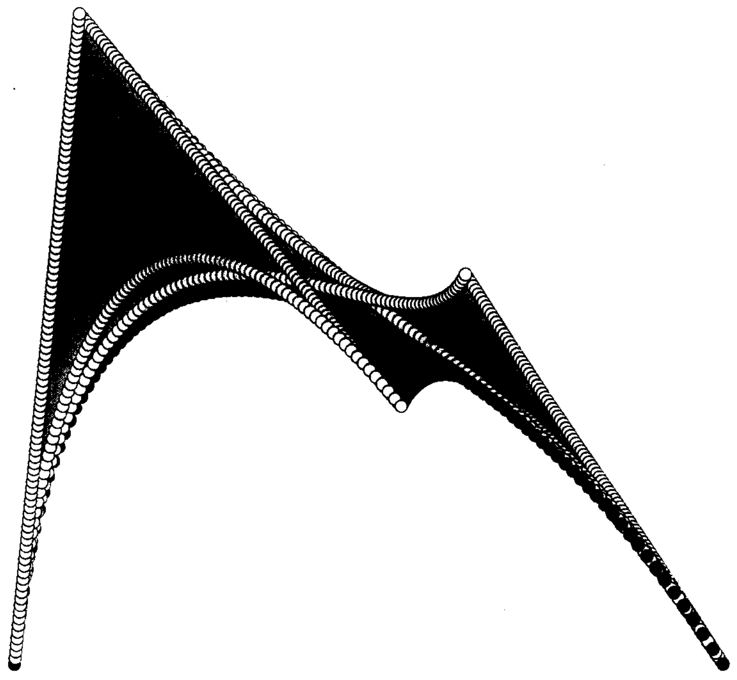

Figure 5.2 The de Casteljau algorithm: 50 points are computed on a quartic curve, and the intermediate points $\mathbf{b}_i^r$ are connected.

5.2 Properties of Bézier Curves

Many of the properties in this section have already appeared in Chapter 4. They were derived using geometric arguments. We shall now rederive several of them, using algebraic arguments. If the same heading is used here as in Chapter 3, the reader should look there for a complete description of the property in question.

Affine invariance. Barycentric combinations are invariant under affine maps. Therefore, (5.5) gives the algebraic verification of this property. We note again that this does not imply invariance under perspective maps!

Invariance under affine parameter transformations. Algebraically, this property reads

$$\sum_{i=0}^{n} \mathbf{b}_i B_i^n(t) = \sum_{i=0}^{n} \mathbf{b}_i B_i^n\left(\frac{u-a}{b-a}\right). \tag{5.9}$$

Convex hull property. This follows, since for $t \in [0,1]$, the Bernstein polynomials are nonnegative. They sum to one as shown in (5.5). For values of t outside $[0,1]$, the convex hull property does not hold; Figure 5.3 illustrates.

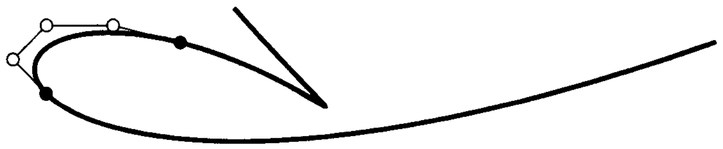

Figure 5.3 Convex hull property: a quartic Bézier curve is plotted for parameter values $t \in [-1, 2]$.

Endpoint interpolation. This is a consequence of the identities

$$\begin{aligned} B_i^n(0) &= \delta_{i,0} \\ B_i^n(1) &= \delta_{i,n} \end{aligned} \tag{5.10}$$

and (5.5). Here, $\delta_{i,j}$ is the Kronecker delta function: it equals one when its arguments agree, and zero otherwise.

Symmetry. Looking at the examples in Figure 4.4, it is clear that it does not matter if the Bézier points are labeled $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n$ or $\mathbf{b}_n, \mathbf{b}_{n-1}, \ldots, \mathbf{b}_0$. The curves that correspond to the two different orderings look the same; they differ only in the direction in which they are traversed. Written as a formula:

$$\sum_{j=0}^{n} \mathbf{b}_j B_j^n(t) = \sum_{j=0}^{n} \mathbf{b}_{n-j} B_j^n(1-t). \tag{5.11}$$

This follows from the identity

$$B_j^n(t) = B_{n-j}^n(1-t), \tag{5.12}$$

which follows from inspection of (5.1). We say that Bernstein polynomials are *symmetric* with respect to t and $1 - t$.

Invariance under barycentric combinations. The process of forming the Bézier curve from the Bézier polygon leaves barycentric combinations invariant. For $\alpha + \beta = 1$, we obtain

$$\sum_{j=0}^{n} (\alpha \mathbf{b}_j + \beta \mathbf{c}_j) B_j^n(t) = \alpha \sum_{j=0}^{n} \mathbf{b}_j B_j^n(t) + \beta \sum_{j=0}^{n} \mathbf{c}_j B_j^n(t). \tag{5.13}$$

In words: we can construct the weighted average of two Bézier curves either by taking the weighted average of corresponding points on the curves, or by taking the weighted average of corresponding control vertices and then computing the curve.

This linearity property is essential for many theoretical purposes, the most important one being the definition of tensor product surfaces in Chapter 14. It is illustrated in Figure 5.4.

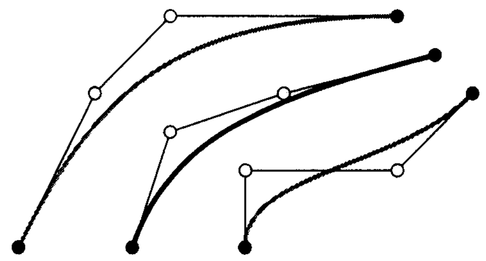

Figure 5.4 Barycentric combinations: the middle curve (black) is the average of the two outer curves (gray).

Linear precision. The following is a useful identity:

$$\sum_{j=0}^{n} \frac{j}{n} B_j^n(t) = t, \tag{5.14}$$

which has the following application: suppose the polygon vertices $\mathbf{b}_j$ are uniformly distributed on a straight line joining two points $\mathbf{p}$ and $\mathbf{q}$:

$$\mathbf{b}_j = \left(1 - \frac{j}{n}\right)\mathbf{p} + \frac{j}{n}\mathbf{q}; \quad j = 0, \ldots, n.$$

The curve that is generated by this polygon is the straight line between $\mathbf{p}$ and $\mathbf{q}$, that is, the initial straight line is reproduced. This property is called *linear precision*.[1]

Pseudolocal control. The Bernstein polynomial B_i^n has only one maximum and attains it at $t = i/n$. This has a design application: if we move only one of the control polygon vertices, say, $\mathbf{b}_i$, then the curve is mostly affected by this change in the region of the curve around the parameter value i/n. This makes the effect of the change reasonably predictable, although the change does affect the whole curve. As a rule of thumb (mentioned to me by P. Bézier), the maximum of each B_i^n is roughly 1/3; thus a change of $\mathbf{b}_i$ by three units will change the curve by one unit.

5.3 The Derivatives of a Bézier Curve

We start with an identity, closely resembling Leibniz's formula for derivatives. Let t be a point on the real line, and let $\vec{v}$ be a vector in the associated 1D linear

1 If the points are not uniformly spaced, we will also recapture the straight line segment. However, it will not be linearly parametrized.

space. Then

$$\mathbf{b}[(t+\vec{v})^{<n>}] = \sum_{i=0}^{n} \binom{n}{i} \mathbf{b}[t^{<n-i>}, \vec{v}^{<i>}]. \tag{5.15}$$

This is an immediate consequence of the Leibniz formula (3.22).

The derivative of a curve $\mathbf{x}(t)$ is typically defined as

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \lim_{h\to 0} \frac{1}{h}[\mathbf{x}(t+h) - \mathbf{x}(t)].$$

We will be a little more precise and observe that t is a 1D point, whereas h is a 1D vector. We thus denote it by $\vec{h}$ and obtain

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \lim_{\vec{h}\to \vec{0}} \frac{1}{|\vec{h}|}[\mathbf{x}(t+\vec{h}) - \mathbf{x}(t)].$$

Invoking (5.15), we have

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \lim_{\vec{h}\to \vec{0}} \frac{1}{|\vec{h}|}\left[\sum_{i=0}^{n} \binom{n}{i} \mathbf{b}[t^{<n-i>}, \vec{h}^{<i>}] - \mathbf{b}[t^{<n>}]\right]. \tag{5.16}$$

For $i = 0$, two terms $\mathbf{b}[t^{<n>}]$ cancel. We expand the rest and factor in the term $|\vec{h}|$:

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \lim_{\vec{h}\to \vec{0}} \left(n\mathbf{b}\left[t^{<n-1>}, \frac{\vec{h}}{|\vec{h}|}\right] + \binom{n}{2}\mathbf{b}\left[t^{<n-2>}, \frac{\vec{h}}{|\vec{h}|}, \vec{h}\right] + \ldots \right)$$

We observe that $\frac{\vec{h}}{|\vec{h}|} = \vec{1}$. Taking the limit annihilates all other terms containing $\vec{h}$, and we thus have

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = n\mathbf{b}[t^{<n-1>}, \vec{1}]. \tag{5.17}$$

Figure 5.5 illustrates the cubic case.

From now on, we use the expression $\dot{\mathbf{x}}(t)$ for the first derivative.

This has two possible interpretations. For the first one, we perform a de Casteljau step with respect to $\vec{1}$, and then $n-1$ steps with respect to t; as an equation:

$$\dot{\mathbf{x}}(t) = n \sum_{j=0}^{n-1} (\mathbf{b}_{j+1} - \mathbf{b}_j) B_j^{n-1}(t).$$

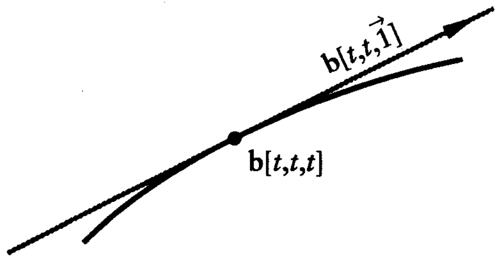

Figure 5.5 Blossoms and derivatives: the underlying geometry.

This can be simplified somewhat by the introduction of the *forward difference operator* Δ:

$$\Delta\mathbf{b}_j = \mathbf{b}_{j+1} - \mathbf{b}_j. \tag{5.18}$$

We now have for the derivative of a Bézier curve:

$$\dot{\mathbf{x}}(t) = n\sum_{j=0}^{n-1} \Delta\mathbf{b}_j B_j^{n-1}(t); \quad \Delta\mathbf{b}_j \in \mathbb{R}^3. \tag{5.19}$$

The derivative of a Bézier curve is thus another Bézier curve, obtained by differencing the original control polygon. However, this derivative Bézier curve does not "live" in $\mathbb{E}^3$ any more! Its coefficients are differences of points, that is, *vectors*, which are elements of $\mathbb{R}^3$. To visualize the derivative curve and polygon in $\mathbb{E}^3$, we can construct a polygon in $\mathbb{E}^3$ that consists of the points $\mathbf{a} + \Delta\mathbf{b}_0, \ldots, \mathbf{a} + \Delta\mathbf{b}_{n-1}$. Here $\mathbf{a}$ is arbitrary; one reasonable choice is $\mathbf{a} = \mathbf{0}$. Figure 5.6 illustrates a Bézier curve and its derivative curve (with the choice $\mathbf{a} = \mathbf{0}$). This derivative curve is sometimes called a *hodograph*. For more information on hodographs, see Forrest [244], Bézier [59], or Sederberg and Wang [559].

For a second interpretation of (5.17), we first perform $n-1$ steps of the de Casteljau algorithm, resulting in the two points $\mathbf{b}_1^{n-1}(t)$ and $\mathbf{b}_0^{n-1}(t)$. Now performing one step with respect to $\vec{1}$ yields (after multiplication by n):

$$\dot{\mathbf{x}}(t) = n\left(\mathbf{b}_1^{n-1}(t) - \mathbf{b}_0^{n-1}(t)\right). \tag{5.20}$$

Thus the first derivative vector is a "byproduct" of the de Casteljau algorithm; see Figure 4.2. The de Casteljau algorithm is not the fastest way to evaluate a Bézier curve, but this property makes it a desirable tool: very often, we not only need a point on a curve, but the derivative vector as well. Using (5.20), we get both in parallel. The two ways of computing the derivative are shown in Example 5.1.

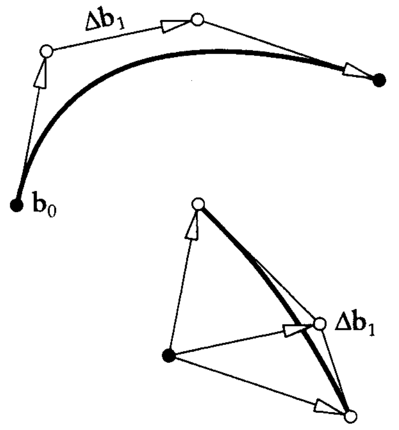

Figure 5.6 Derivatives: a Bézier curve and its first derivative curve (scaled down by a factor of three). Note that this derivative curve does not change if a translation is applied to the original curve.

Example 5.1 **Two ways to compute derivatives.**

To compute the derivative of the Bézier curve from Example 4.1, we could form the first differences of the control points and evaluate the corresponding quadratic curve at $t = 1/2$:

$$\begin{array}{ccc} \begin{bmatrix}0\\2\end{bmatrix} & & \\ \begin{bmatrix}8\\0\end{bmatrix} & \begin{bmatrix}4\\1\end{bmatrix} & \\ \begin{bmatrix}-4\\-2\end{bmatrix} & \begin{bmatrix}2\\-1\end{bmatrix} & \begin{bmatrix}3\\0\end{bmatrix} \end{array}$$

Alternatively, we could compute the difference $\mathbf{b}_1^2 - \mathbf{b}_0^2$:

$$\begin{bmatrix}5\\ \frac{3}{2}\end{bmatrix} - \begin{bmatrix}2\\ \frac{3}{2}\end{bmatrix} = \begin{bmatrix}3\\0\end{bmatrix}.$$

In both cases, the result needs to be multiplied by a factor of 3.

Higher derivatives follow the same pattern:

$$\frac{\mathrm{d}^r \mathbf{x}(t)}{\mathrm{d}t^r} = \frac{n}{(n-r)!}\mathbf{b}[t^{<n-r>}, \vec{1}^{<r>}]. \tag{5.21}$$

To compute these derivatives from the Bézier points, we first generalize the forward difference operator (5.18): the *iterated forward difference operator* Δ^r is defined by

$$\Delta^r \mathbf{b}_j = \Delta^{r-1}\mathbf{b}_{j+1} - \Delta^{r-1}\mathbf{b}_j. \tag{5.22}$$

We list a few examples:

$$\Delta^0 \mathbf{b}_i = \mathbf{b}_i$$
$$\Delta^1 \mathbf{b}_i = \mathbf{b}_{i+1} - \mathbf{b}_i$$
$$\Delta^2 \mathbf{b}_i = \mathbf{b}_{i+2} - 2\mathbf{b}_{i+1} + \mathbf{b}_i$$
$$\Delta^3 \mathbf{b}_i = \mathbf{b}_{i+3} - 3\mathbf{b}_{i+2} + 3\mathbf{b}_{i+1} - \mathbf{b}_i.$$

The factors on the right-hand sides are binomial coefficients, forming a Pascal-like triangle. This pattern holds in general:

$$\Delta^r \mathbf{b}_i = \sum_{j=0}^{r} \binom{r}{j} (-1)^{r-j} \mathbf{b}_{i+j}. \tag{5.23}$$

The r^{th} derivative of a Bézier curve is now given by

$$\frac{\mathrm{d}^r}{\mathrm{d}t^r}\mathbf{b}^n(t) = \frac{n!}{(n-r)!}\sum_{j=0}^{n-r} \Delta^r \mathbf{b}_j B_j^{n-r}(t). \tag{5.24}$$

Two important special cases of (5.24) are given by $t = 0$ and $t = 1$. Because of (5.10), we obtain

$$\frac{\mathrm{d}^r}{\mathrm{d}t^r}\mathbf{b}^n(0) = \frac{n!}{(n-r)!}\Delta^r \mathbf{b}_0 \tag{5.25}$$

and

$$\frac{\mathrm{d}^r}{\mathrm{d}t^r}\mathbf{b}^n(1) = \frac{n!}{(n-r)!}\Delta^r \mathbf{b}_{n-r}. \tag{5.26}$$

Thus the r^{th} derivative of a Bézier curve at an endpoint depends only on the $r + 1$ Bézier points near (and including) that endpoint. For $r = 0$, we get the already established property of endpoint interpolation. The case $r = 1$ states that $\mathbf{b}_0$ and

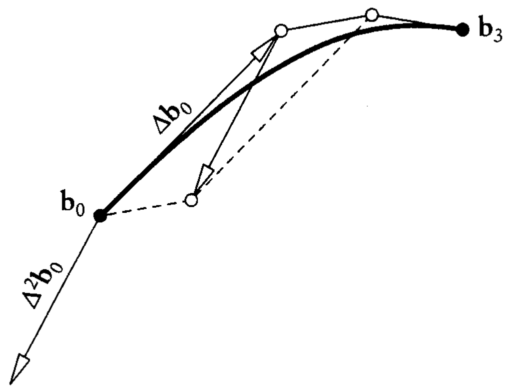

Figure 5.7 Endpoint derivatives: the first and second derivative vectors at $t = 0$ are multiples of the first and second difference vectors at $\mathbf{b}_0$.

$\mathbf{b}_1$ define the tangent at $t = 0$, provided they are distinct.[2] Similarly, $\mathbf{b}_{n-1}$ and $\mathbf{b}_n$ determine the tangent at $t = 1$. The cases $r = 1$, $r = 2$ are illustrated in Figure 5.7.

If we know all derivatives of a function at one point, corresponding to $t = 0$, say, we can generate its Taylor series. The Taylor series of a polynomial is just that polynomial itself, in the *monomial form*:

$$\mathbf{x}(t) = \sum_{j=0}^{n} \frac{1}{j!}\mathbf{x}^{(j)}(0)t^j.$$

Using (5.25), we have

$$\mathbf{b}^n(t) = \sum_{j=0}^{n} \binom{n}{j} \Delta^j \mathbf{b}_0 \, t^j. \tag{5.27}$$

The monomial form should be avoided wherever possible; it is very unstable for floating-point operations.

If $\mathbf{x}(t)$ is defined over an interval $[a, b]$, (5.17) becomes

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \frac{n}{b-a}\mathbf{b}[t^{<n-1>}, \vec{1}]. \tag{5.28}$$

2 In general, the tangent at $\mathbf{b}_0$ is determined by $\mathbf{b}_0$ and the first $\mathbf{b}_i$ that is distinct from $\mathbf{b}_0$. Thus the tangent may be defined even if the tangent vector is the zero vector.

5.4 Domain Changes and Subdivision

A Bézier curve $\mathbf{b}^n$ is usually defined over the interval (the domain) $[0,1]$, but it can also be defined over any interval $[0,c]$. The part of the curve that corresponds to $[0,c]$ can also be defined by a Bézier polygon, as illustrated in Figure 5.8. Finding this Bézier polygon is referred to as *subdivision* of the Bézier curve.

The unknown Bézier points $\mathbf{c}_i$ are found without much work if we use the blossoming principle from Section 4.4. There, (4.11) gave us the Bézier points of a polynomial curve that is defined over an arbitrary interval $[a,b]$. We are currently interested in the interval $[0,c]$, and so our Bézier points are:

$$\mathbf{c}_i = \mathbf{b}[0^{<n-i>}, c^{<i>}].$$

Thus each $\mathbf{c}_i$ is obtained by carrying out i de Casteljau steps with respect to c, in nonblossom notation:

$$\mathbf{c}_j = \mathbf{b}_0^j(c). \tag{5.29}$$

This formula is called the *subdivision formula* for Bézier curves.

Thus it turns out that the de Casteljau algorithm not only computes the point $\mathbf{b}^n(c)$, but also provides the control vertices of the Bézier curve corresponding to the interval $[0,c]$. Because of the symmetry property (5.11), it follows that the control vertices of the part corresponding to $[c,1]$ are given by the $\mathbf{b}_j^{n-j}$. Thus, in Figures 4.1 and 4.2, we see the two subpolygons defining the arcs from $\mathbf{b}^n(0)$ to $\mathbf{b}^n(t)$ and from $\mathbf{b}^n(t)$ to $\mathbf{b}^n(1)$.

Instead of subdividing a Bézier curve, we may also *extrapolate* it: in that case, we might be interested in the Bézier points $\mathbf{d}_i$ corresponding to an interval $[1,d]$. They are given by

$$\mathbf{d}_j = \mathbf{b}[1^{<n-j>}, d^{<j>}] = \mathbf{b}_{n-j}^j(d).$$

It should be mentioned that extrapolation is not a numerically stable process, and should be avoided for large values of d.

Subdivision for Bézier curves, although mentioned by de Casteljau [146], was rigorously proved by E. Staerk [578]. Our blossom development is due to Ramshaw [498] and de Casteljau [147].

Subdivision may be repeated: we may subdivide a curve at $t = 1/2$, then split the two resulting curves at $t = 1/2$ of their respective parameters, and so on. After k levels of subdivisions, we end up with 2^k Bézier polygons, each describing a small arc of the original curve. These polygons converge to the curve if we keep increasing k, as was shown by Lane and Riesenfeld [369].

Convergence of this repeated subdivision process is very fast (see Cohen and Schumaker [123] and Dahmen [131]), and thus it has many practical applica-

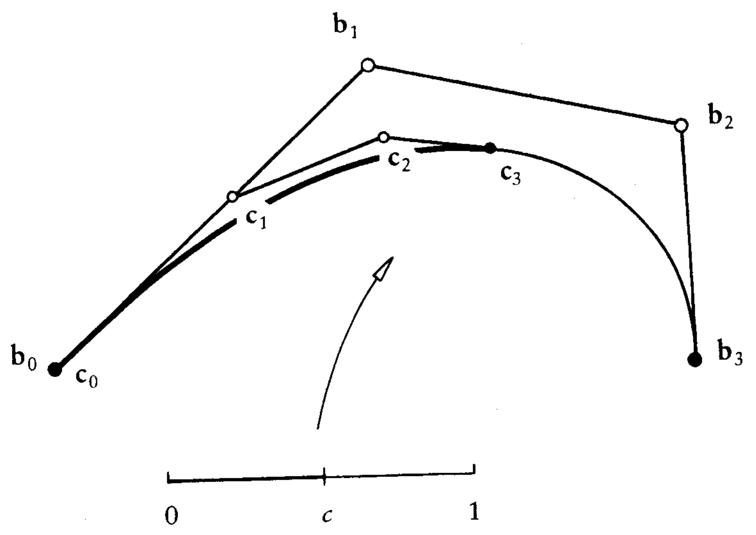

Figure 5.8 Subdivision: two Bézier polygons describing the same curve: one (the $\mathbf{b}_i$) is associated with the parameter interval [0, 1], the other (the $\mathbf{c}_i$) with $[0, c]$.

tions. We shall discuss here the process of intersecting a straight line with a Bézier curve. Suppose we are given a planar Bézier curve and we wish to find intersection points with a given straight line L, if they exist.

If the curve and L are far apart, we would like to be able to flag such configurations as quickly as possible, and then abandon any further attempts to find intersection points. To do this, we create the *minmax box* of the control polygon: this is the smallest rectangle, with sides parallel to the coordinate axes, that contains the polygon. It is found very quickly, and by the convex hull property of Bézier curves, we know that it also contains the curve. Figure 5.9 gives an example.

Having found the minmax box, it is trivial to determine if it interferes with L; if not, we know we will not have any intersections. This quick test is called *trivial reject*.

Now suppose the minmax box *does* interfere with L. Then there may be an intersection. We now subdivide the curve at $t = 1/2$ and carry out our trivial reject test for both subpolygons.[3] If the outcome is still inconclusive, we repeat. Eventually the size of the involved minmax boxes will be so small that we can simply take their centers as the desired intersection points.

The routine intersect employs this idea, and a little more: as we keep subdividing the curve, zooming in toward the intersection points, the generated subpolygons become simpler and simpler in shape. If the control points of a

3 The choice $t = 1/2$ is arbitrary, but works well. We might try to find better places to subdivide, but it is cheaper to just perform a few more subdivisions instead.

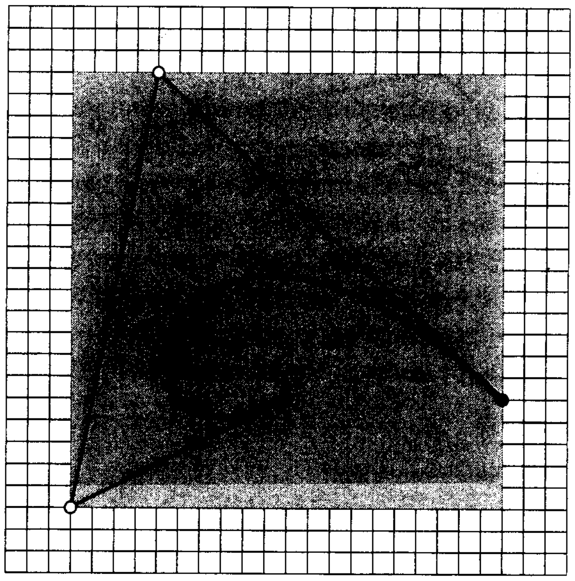

Figure 5.9 The minmax box of a Bézier curve: the smallest rectangle that contains the curve's control polygon.

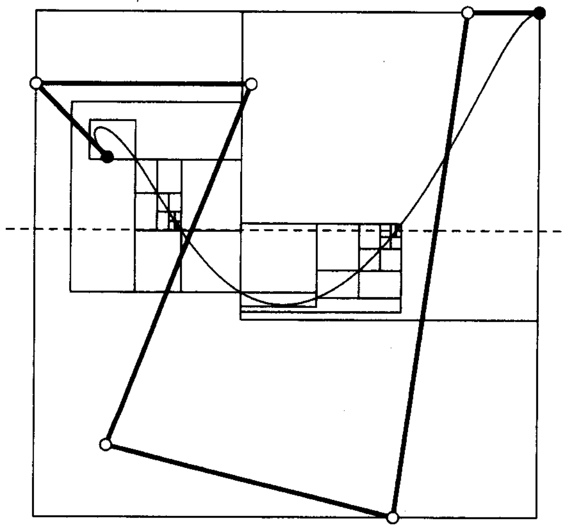

Figure 5.10 Subdivision: finding the intersections of a curve with a line (dashed). Note the clustering of minmax boxes near the intersection points.

polygon are almost collinear, we may replace them with a straight line. We could then intersect this straight line with L in order to find an intersection point. The extra work here lies in determining if a control polygon is "linear" or not. In our case, this is done by the routine `checkflat`. Figure 5.10 gives an example. Note how the subdivision process finds *all* intersection points. These points will not, however, be recorded by increasing values of t.

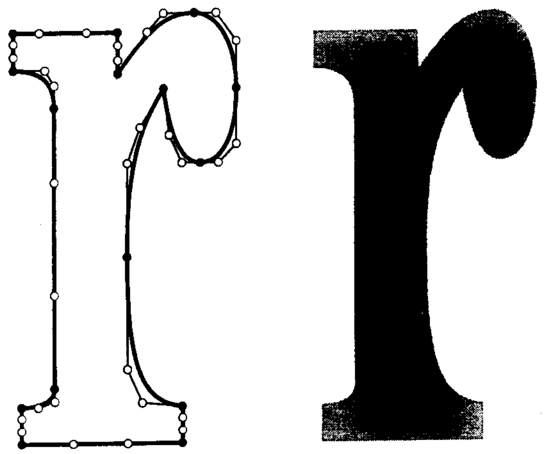

Figure 5.11 Font design: the characters in this book are stored as a sequence of cubic Bézier curves.

5.5 **Composite Bézier Curves**

Curves may be composed of several Bézier curves in order to generate shapes that are too complex for a single Bézier curve to handle. For example, Figure 5.11 shows how composite Bézier curves may be used in *font design*.[4]

In piecing Bézier curves together, we need to control the smoothness of the resulting curve. Let $\mathbf{b}_0, \ldots, \mathbf{b}_3$ and $\mathbf{b}_3, \ldots, \mathbf{b}_6$ be the Bézier points of two cubic curve segments $\mathbf{x}_-$ and $\mathbf{x}_+$. Since they both share the point $\mathbf{b}_3$, they clearly form a continuous, or C^0, curve. With this minimal continuity requirement, the two curves may form a corner; for several examples, see Figure 5.11.

But if we want to ensure that the two pieces meet smoothly, more care is called for. Based on our knowledge of endpoint derivatives from Section 5.3, the three points $\mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4$ must be collinear. That condition ensures that the tangent[5] at $\mathbf{b}_3$ is the same for both curves. Again, consult Figure 5.11 for examples. Curves with a continuously changing tangent are called G^1, or first-order geometrically continuous; see Chapter 11.

A stronger condition is to require that the two curve segments form a C^1, or continuously differentiable curve. Since the derivative of a curve (more precisely, the length of the derivative vector) depends on the domain of the curve, we need to introduce domains for our two curve segments. We adopt the convention that $\mathbf{x}_-$ is defined over an interval $[a, b]$ and that $\mathbf{x}_+$ is defined over $[b, c]$. The derivatives

4 This book was printed using the PostScript language. It represents all characters as piecewise cubic Bézier curves in order to have a *scalable* font set. As an estimate, the text in this book is made up using about 10 million cubic Bézier curves.

5 By "tangent," we refer to the tangent line, not to the derivative vector!

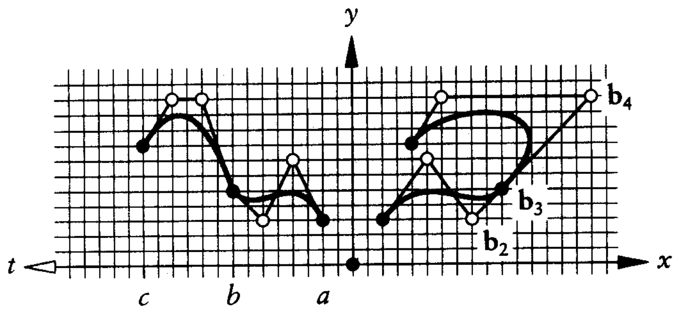

Figure 5.12 Composite curves: a C^0 example.

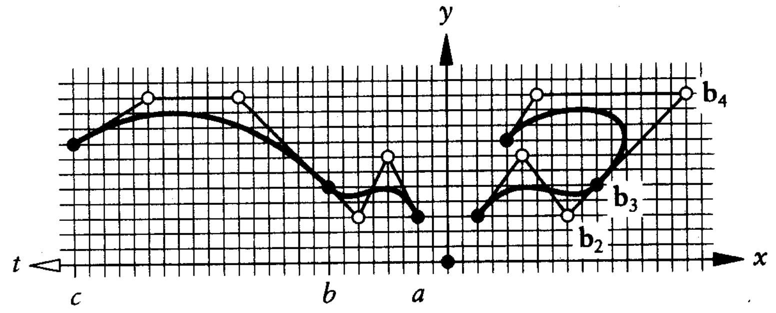

Figure 5.13 Composite curves: a C^1 example.

of both segments at parameter value b are now obtained using (5.28):

$$\frac{3}{b-a}[\mathbf{b}_3 - \mathbf{b}_2] = \frac{3}{c-b}[\mathbf{b}_4 - \mathbf{b}_3]. \tag{5.30}$$

A geometric interpretation is that the ratio of the three points $\mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4$ is the same as the ratio of the three parameter values a, b, c. This is a much stronger condition than that for G^1 continuity above!

Figures 5.12 and 5.13 illustrate this difference. The composite parametric curves—in the x, y-coordinate systems—are identical. The difference is their domains: in Figure 5.12, we chose $a, b, c = 0, 1, 2$. Thus ratio$(a, b, c) = 1$ while the figure suggests that ratio$(\mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4) = 1/3$. Hence the composite curve is not C^1, despite the collinearity of the points $\mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4$. This is demonstrated by the cross plot (y-part only): each component must form a C^1 function for a curve to be C^1. Clearly, the y-component is not C^1.

If we adjust the domain, however, such that the range geometry is reflected by the domain geometry, we can achieve C^1. This is shown in Figure 5.13, where now ratio$(a, b, c) = 1/3$. This results in C^1 components, and hence also in a C^1 composite curve.

Higher-order smoothness of composite curves is best dealt with in the context of B-spline curves and blossoms; see Section 8.7.

5.6 Blossom and Polar

After the first de Casteljau step with respect to a parameter value t_1, the resulting $\mathbf{b}_0^1(t_1), \ldots, \mathbf{b}_{n-1}^1(t_1)$ may be interpreted as a control polygon of a curve $\mathbf{p}_1(t)$ of degree $n-1$. In the blossoming terminology from Section 4.4, we can write:

$$\mathbf{p}_1(t) = \mathbf{b}[t_1, t^{<n-1>}].$$

Invoking our knowledge about derivatives, we have:

$$\begin{aligned}
\mathbf{p}_1(t) &= \sum_{i=0}^{n-1} \left[(1-t_1)\mathbf{b}_i + t_1\mathbf{b}_{i+1}\right] B_i^{n-1}(t) \\
&= \sum_{i=0}^{n-1} \left[(1-t_1)\mathbf{b}_i + t_1\mathbf{b}_{i+1} - \mathbf{b}_i^1(t)\right] B_i^{n-1}(t) + \sum_{i=0}^{n-1} \mathbf{b}_i^1(t) B_i^{n-1}(t) \\
&= (t_1 - t)\sum_{i=0}^{n-1} [\mathbf{b}_{i+1} - \mathbf{b}_i] B_i^{n-1}(t) + \sum_{i=0}^{n-1} \mathbf{b}_i^1(t) B_i^{n-1}(t).
\end{aligned}$$

Therefore,

$$\mathbf{p}_1(t) = \mathbf{b}(t) + \frac{t_1 - t}{n}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{b}(t). \tag{5.31}$$

The polynomial $\mathbf{p}_1$ is called *first polar* of $\mathbf{b}(t)$ with respect to t_1. Figure 5.14 illustrates the geometric significance of (5.31): the tangent at any point $\mathbf{b}(t)$ intersects the polar at $\mathbf{p}_1(t)$. Keep in mind that this is not restricted to planar curves, but is equally valid for space curves!

For the special case of a (nonplanar) cubic, we may then conclude the following: the polar $\mathbf{p}_1$ lies in the osculating plane (see Section 11.2) of the cubic at $\mathbf{b}(t_1)$. If we intersect all tangents to the cubic with this osculating plane, we will trace out the polar. We can also conclude that for three different parameters t_1, t_2, t_3, the blossom value $\mathbf{b}[t_1, t_2, t_3]$ is the intersection of the corresponding osculating planes.

Another special case is given by $\mathbf{b}[0, t^{<n-1>}]$: this is the polynomial defined by $\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}$. Similarly, $\mathbf{b}[1, t^{<n-1>}]$ is defined by $\mathbf{b}_1, \ldots, \mathbf{b}_n$. This observation may be used for a proof of (4.9).

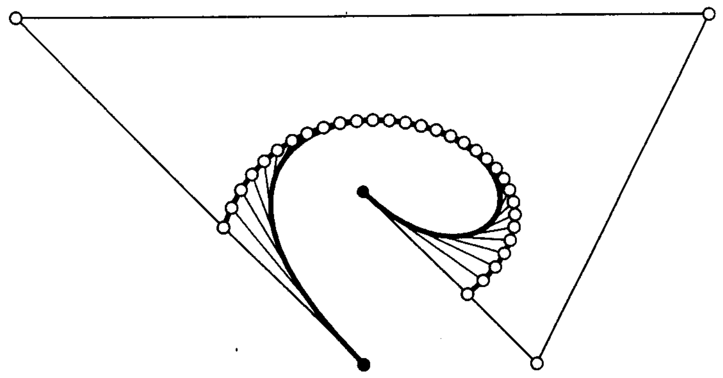

Figure 5.14 Polars: the polar $\mathbf{p}_1(t)$ with respect to $t_1 = 0.4$ is intersected by the tangents of the given curve $\mathbf{b}(t)$.

Returning to the general case, we may repeat the process of forming polars, thus obtaining a second polar $\mathbf{p}_{1,2}(t) = \mathbf{b}[t_1, t_2, t^{<n-2>}]$, and so on. We finally arrive at the n^{th} polar, which we have already encountered as the blossom $\mathbf{b}[t_1, \ldots, t_n]$ of $\mathbf{b}(t)$. The relationship between blossoms and polars was observed by Ramshaw in [499]. The preceding geometric arguments are due to S. Jolles, who developed a geometric theory of blossoming as early as 1886 in [346].[6]

5.7 The Matrix Form of a Bézier Curve

Some authors (Faux and Pratt [228], Mortenson [433], Chang [106]) prefer to write Bézier curves and other polynomial curves in matrix form. A curve of the form

$$\mathbf{x}(t) = \sum_{j=0}^{n} \mathbf{c}_i C_i(t)$$

can be interpreted as a dot product:

$$\mathbf{x}(t) = [\,\mathbf{c}_0 \quad \ldots \quad \mathbf{c}_n\,] \begin{bmatrix} C_0(t) \\ \vdots \\ C_n(t) \end{bmatrix}.$$

One can take this a step further and write

6 W. Boehm first noted the relevance of Jolles's work to the theory of blossoming.

$$\begin{bmatrix} C_0(t) \\ \vdots \\ C_n(t) \end{bmatrix} = \begin{bmatrix} m_{00} & \dots & m_{0n} \\ \vdots & & \vdots \\ m_{n0} & \dots & m_{nn} \end{bmatrix} \begin{bmatrix} t^0 \\ \vdots \\ t^n \end{bmatrix}. \tag{5.32}$$

The matrix $M = \{m_{ij}\}$ describes the basis transformation between the basis polynomials $C_i(t)$ and the *monomial basis* t^i.

If the C_i are Bernstein polynomials, $C_i = B_i^n$, the matrix M has elements

$$m_{ij} = (-1)^{j-i}\binom{n}{j}\binom{j}{i}, \tag{5.33}$$

a simple consequence of (5.27).

We list the cubic case explicitly:

$$M = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The matrix form (5.32) does not deseribe an actual Bézier curve; it is rather the monomial form, which is *numerically unstable* and should be avoided where accuracy in computation is of any importance. See the discussion in Section 24.3 for more details.

5.8 Implementation

First, we provide a routine that evaluates a Bézier curve more efficiently than decas from the last chapter. It will have the flavor of Horner's scheme for the evaluation of a polynomial in monomial form. To give an example of Horner's scheme, also called *nested multiplication*, we list the cubic case:

$$\mathbf{c}_0 + t\mathbf{c}_1 + t^2\mathbf{c}_2 + t^3\mathbf{c}_3 = \mathbf{c}_0 + t[\mathbf{c}_1 + t(\mathbf{c}_2 + t\mathbf{c}_3)].$$

A similar nested form can be devised for Bézier curves; again, the cubic case:

$$\mathbf{b}^3(t) = \left\{\left[\binom{3}{0}s\mathbf{b}_0 + \binom{3}{1}t\mathbf{b}_1\right]s + \binom{3}{2}t^2\mathbf{b}_2\right\}s + \binom{3}{3}t^3\mathbf{b}_3,$$

where $s = 1 - t$. Recalling the identity

$$\binom{n}{i} = \frac{n-i+1}{i}\binom{n}{i-1}; \quad i > 0,$$

we arrive at the following program (for the general case):

```
float hornbez(degree,coeff,t)
/*   uses  a Horner-like  scheme 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.
*/
```

To use this routine for plotting a Bézier curve, we would replace the call to decas in bez_to_points by an identical call to hornbez. Replacing decas with hornbez results in a significant savings of time: we do not have to save the control polygon in an auxiliary array; also, hornbez is of order n, whereas decas is of order n^2.

This is not to say, however, that we have produced superefficient code for plotting points on a Bézier curve. For instance, we have to call hornbez once for each coordinate, and thus have to generate the binomial coefficients n_choose_i twice. This could be improved by writing a routine that combines the two calls. A further improvement could be to compute the sequence of binomial coefficients only once, and not over and over for each new value of t. All these (and possibly more) improvements would speed up the program, but would be less modular and thus less understandable. For the code in this book, modularity is placed above efficiency (in most cases).

We also include the programs to convert from the Bézier form to the monomial form:

```
void bezier_to_power(degree,bez,coeff)
/*Converts Bezier form to power (monomial) form. Works on
one coordinate only.

  Input:   degree:    degree of curve.
           bez:       coefficients of Bezier form
  Output:  coeff:     coefficients of power form.

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

The conversion program internally calls iterated forward differences:

```
void differences(degree,coeff,diffs)
/*
Computes all forward differences Delta^i(b_0).
Has to be called for each coordinate  (x,y, and/or z) of a control polygon.
    Input:   degree: length (from 0) of coeff.
             coeff:  array of coefficients.
     Output: diffs:  diffs[i]= Delta^i(coeff[0]).
*/
```

Once the power form is found, it may be evaluated using Horner's scheme:

```
float horner(degree,coeff,t)
/*
     uses  Horner's scheme to compute one coordinate
     value of a curve in power form. 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.
*/
```

The subdivision routine:

```
void subdiv(degree,coeff,weight,t,bleft,bright,wleft,wright)
/*
       subdivides ratbez curve at parameter value t.
  Input:  degree:    degree of Bezier curve
          coeff:     Bezier points (one coordinate only)
          weight:    weights for rational case
          t:         where to subdivide
  Output:
          bleft,bright: left and right subpolygons
          wleft,wright: their weights


  Note:   1. For the polynomial case, set all entries in weight to 1.
          2. Ordering of right polygon bright is reversed.
*/
```

Actually, this routine computes a more general case than is described in this chapter; namely, it computes subdivison for a *rational* Bézier curve. This will be discussed later; if the entries in `weight` are all unity, then `wleft` and `wright` will also be unity and can be safely ignored in the context of this chapter.

Now we present the routine to intersect a Bézier curve with a straight line (the straight line is assumed to be the x-axis):

```
void intersect(bx,by,w,degree,tol)
/* Intersects Bezier curve with x-axis  by  adaptive subdivision.
   Subdivision is controlled by tolerance tol. There is
   no check for stack depth! Intersection points are not found  in
  'natural' order.  Results are written into file outfile.
 Input: bx,by,w:     rational Bezier curve
        degree:      its degree
        tol:         accuracy for results
 Output:             intersection points, written into a file
 */
```

This routine (again covering the rational case as well) uses a routine to check if a control polygon is flat:

```
int check_flat(bx,by,degree,tol)
/* Checks if a polygon is flat. If all points
   are closer  than tol to the connection of the
   two endpoints, then it is flat. Crashes if the endpoints
   are identical.

  Input:   bx,by, degree: the Bezier curve
           tol:           tolerance
  Output:  1 if flat, 0 else.
*/
```

5.9 Problems

1 Consider the cubic Bézier curve given by the planar control points

$$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}-1\\0\end{bmatrix}.$$

At $t = 1/2$, this curve has a *cusp*: its first derivative vanishes and it shows a sharp corner. You should verify this by a sketch. Now perturb the x-coordinates of $\mathbf{b}_1$ and $\mathbf{b}_2$ by opposite amounts, thus maintaining a symmetric control polygon. Discuss what happens to the curve.

2 Show that a nonplanar cubic Bézier curve cannot have a cusp. Hint: use the fact that $\mathbf{b}_0^{n-1}, \mathbf{b}_1^{n-1}, \mathbf{b}_0^n$ are identical when we evaluate at the cusp.

3 Show that the Bernstein polynomial B_i^n attains its maximum at $t = i/n$. Find the maximum value. What happens for large n?

***4** Show that the Bernstein polynomials B_i^n form a basis for the linear space of all polynomials of degree n.

P1 Compare the run times of decas and hornbez for curves of various degrees.

P2 Use subdivision to create *smooth fractals*. Start with a degree four Bézier curve. Subdivide it into two curves and then perturb the middle control point $\mathbf{b}_2$ for each of the two subpolygons. Continue for several levels. Try to perturb the middle control point by a random displacement and then by a controlled displacement. Literature on fractals: [35], [411].

P3 Use subdivision to approximate a high-order ($n > 2$) Bézier curve by a collection of quadratic Bézier curves. You will have to write a routine that determines if a given Bézier curve may be replaced by a quadratic one within a given tolerance. Literature on approximating higher-order curves by lower-order ones: [336], [341].