Chapter 9

Polynomial Root Finding in Bernstein Form

The Bernstein polynomial basis enjoys widespread use in the fields of CAGD and computer graphics. The use of the Bernstein basis functions to express Bézier curves and surfaces allows many basic algorithms concerned with the processing of such forms to be reduced to the problem of computing the real roots of polynomials in Bernstein form.

A typical example is the problem of computing the intersection points of two Bézier curves using the implicitization algorithm. Given two curves of degrees m and n, respectively, the problem of identifying their intersection points can be reformulated in terms of finding the real roots on the unit interval of a degree mn polynomial in Bernstein form [SP86a]. Other examples from CAGD include finding a point on each loop of the intersection curve of two surfaces and the "trimming" of offset and bisector curves. Examples from computer graphics include ray tracing and algebraic surface rendering.

In the early years of computer graphics and CAGD, the tendency was to convert from the Bernstein to the power basis for root finding since the power basis is more "familiar" and library routines are commonly available to solve for polynomial roots in the power basis. However, it has subsequently been demonstrated [FR87] that the Bernstein basis is inherently better conditioned than the power basis for finding real roots on the unit interval (and this is all that is of interest when dealing with Bézier curves anyway). Thus, root finding algorithms for polynomials in Bernstein form are not merely a convenience, they in fact offer significantly better accuracy than their power basis counterparts.

Furthermore, by expressing Bernstein form polynomials as explicit Bézier curves, one can take advantage of geometric insights based on the control polygon in developing root–finding algorithms. This can allow us to robustly find all real roots in the unit interval, typically much faster than can a library polynomial root finder which finds all real *and* complex roots. Thus, a root finder that works in the Bernstein basis could provide enhanced speed and accuracy when applied to problems whose solution traditionally relies on the computation of real polynomial roots in the power basis.

A detailed study of root finding algorithms for polynomials in Bernstein form is found in [Spe94]. In this chapter we review two basic approaches. The fundamental idea it to express a polynomial as an explicit Bézier curve (Section 2.14).

9.1 Convex Hull Marching

The *Bernstein convex hull approximating step algorithm* finds each real root of $[0, 1]$ in ascending order by computing a sequence of subdivided explicit Bézier curves which are guaranteed to not skip over a real root. Each step is determined by exploiting the convex hull property which assures that all roots lie within the interval where the convex hull of the control polygon intersects the t-axis. A root is realized when its respective sequence of approximating steps converge to a limit.

This algorithm is quite similar to Newton's method, but it has the advantage that it always approaches the root from the left, and hence it cannot diverge like Newton's method.

This algorithm was first overviewed in [SP86a]. Since then, it has been adopted to various applications and to higher dimensions under a technique referred to as Bézier clipping [SWZ89, Sed89, SN90, NSK90].

The algorithm is illustrated in Figure 9.1. Begin by determining the left-most intersection between the convex-hull of the explicit Bézier curve $f_{[0,1]}(t)$. and the t axis. No roots can exist in the interval $[0, t_1]$ in Figure 9.1.a. Next, perform the de Casteljau algorithm (Figure 9.1.b) yielding $f_{[t_1,1]}(t)$ and find the value t_2 where the new convex hull intersects the t axis. These two operations are repeated until $t_{k+1} = t_k$ to within floating point tolerance, and t_k is taken to be a root. In this case, $k = 6$. To within floating point tolerance, $f_0 = 0$ in the explicit Bézier curve $f_{[t_6,1]}(t)$.

One pitfall to be aware of is that, due to floating point roundoff, it is possible (although rare) for the algorithm to sneak past a root. To avoid this problem, you should check to make sure that, at each iteration, $f_{[t_i,1]}(t_i)$ and $f_{[t_{i+1},1]}(t_{i+1})$ have the same sign. If they do not, t_i and t_{i+1} will differ only by floating point error. You should take $f_{[t_i,1]}(t_i)$ to have a root at t_i and proceed with the deflation step.

When a root is found, it is "deflated" out and the algorithm proceeds as before to find the next root. Deflation is the process of finding a degree $n-1$ polynomial $f^1_{[t_k,1]}(t)$which has the same roots as $f_{[t_k,1]}(t)$ except for the root at t_k. If we denote the deflated polynomial by $f^1_{[t_k,1]}(t)$, then

$$f_{[t_k,1]}(t) = (t - t_k) f^1_{[t_k,1]}(t).$$

The coefficients of $f^1_{[t_k,1]}(t)$ are

$$y^1_i = \frac{n}{i+1} y_{i+1}, \qquad i = 0, \ldots, n-1.$$

This can be seen from the following derivation:

$$\begin{aligned}
f_{[0,1]}(t) &= \sum_{i=1}^{n} y_i B_i^n(t) \\
&= \sum_{i=1}^{n} y_i \binom{n}{i} (1-t)^{n-i} t^i \\
&= t \sum_{i=0}^{n-1} y_{i+1} \binom{n}{i+1} (1-t)^{n-1-i} t^i \\
&= t \sum_{i=0}^{n-1} \frac{\binom{n}{i+1}}{\binom{n-1}{i}} y_{i+1} B_i^{n-1} \\
&= t \sum_{i=0}^{n-1} \frac{n}{i+1} y_{i+1} B_i^{n-1}
\end{aligned} \tag{9.1}$$

This algorithm performs reliably. It is numerically stable. The main drawback is the expense of doing a de Casteljau subdivision at each step.

9.2 Bernstein Combined Subdivide & Derivative Algorithm

In general, the most efficient root finding algorithm for polynomials of degree larger than seven is one which isolates roots and then refines them using the modified regula falsi method. A root is isolated if there is exactly one sign change in the Bernstein coefficients (this is due to the variation diminishing property).

The root isolation heuristic is illustrated in Figures 9.2 and 9.3. The basic idea is to perform a binary search to isolate the left-most root. If the root is not isloated after a few de Casteljau subdivisions (first at t=.5, then t=.25, then t=.125), the algorithm computes the left-most root of the derivative polynomial and uses that value to isolate the left-most root of the polynomial.

Figure 9.2.a shows an initial degree five polynomial with simple roots at 0, 0.2, and 1, along with a double root at 0.6. Figure 9.2.b shows the polynomial after deflating the roots at 0 and 1. In Figure 9.2.c, the polynomial has been split in two pieces at τ_1 and the segment $\mathbf{P}^{(2)}_{[0,\tau_1]}(t)$ is determined to contain exactly one root since its control polygon crosses the t axis exactly once. In Figure 9.2.d, $\mathbf{P}^{(3)}_{[\tau_1,1]}(t)$ does not pass the isolation test because its control polygon crosses the t axis twice.

In Figure 9.3.e, $\mathbf{P}^{(3)}_{[\tau_1,1]}(t)$ is split and the right half is seen to have no roots, while the left half is indeterminate. Figure 9.3.f performs another subdivision, with the same results, as does Figure 9.3.g. At this point, the algorithm changes its heuristic for isolating the root from binary search to finding the left–most root of the derivative of $\mathbf{P}^{(6)}(t)$. That derivative, shown in Figure 9.3.h, has only one root in the interval. The root is refined and used to isolate the left–most root of $\mathbf{P}^{(6)}(t)$. In this case, the double root at 0.6 is identified by checking that the numerical value of $\mathbf{P}^{(6)}(0.6)$ is less than the computed running error bound.

This algorithm is generally faster than the one in Section 9.1 because the modified regula falsi refinement algorithm can be preformed using the $O(n)$ Horner's evaluation algorithm instead of the $O(n^2)$ de Casteljau algorithm.

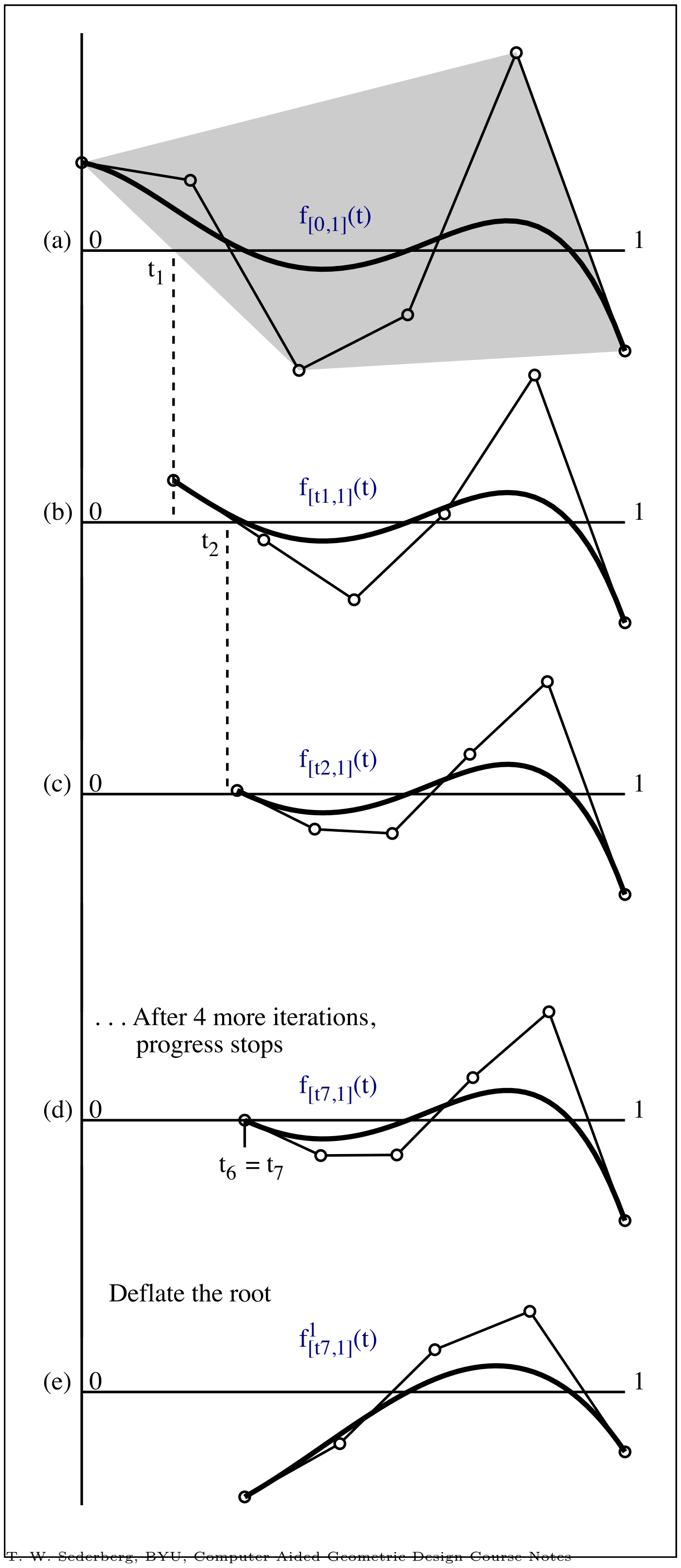

Figure 9.1: Bernstein root finding

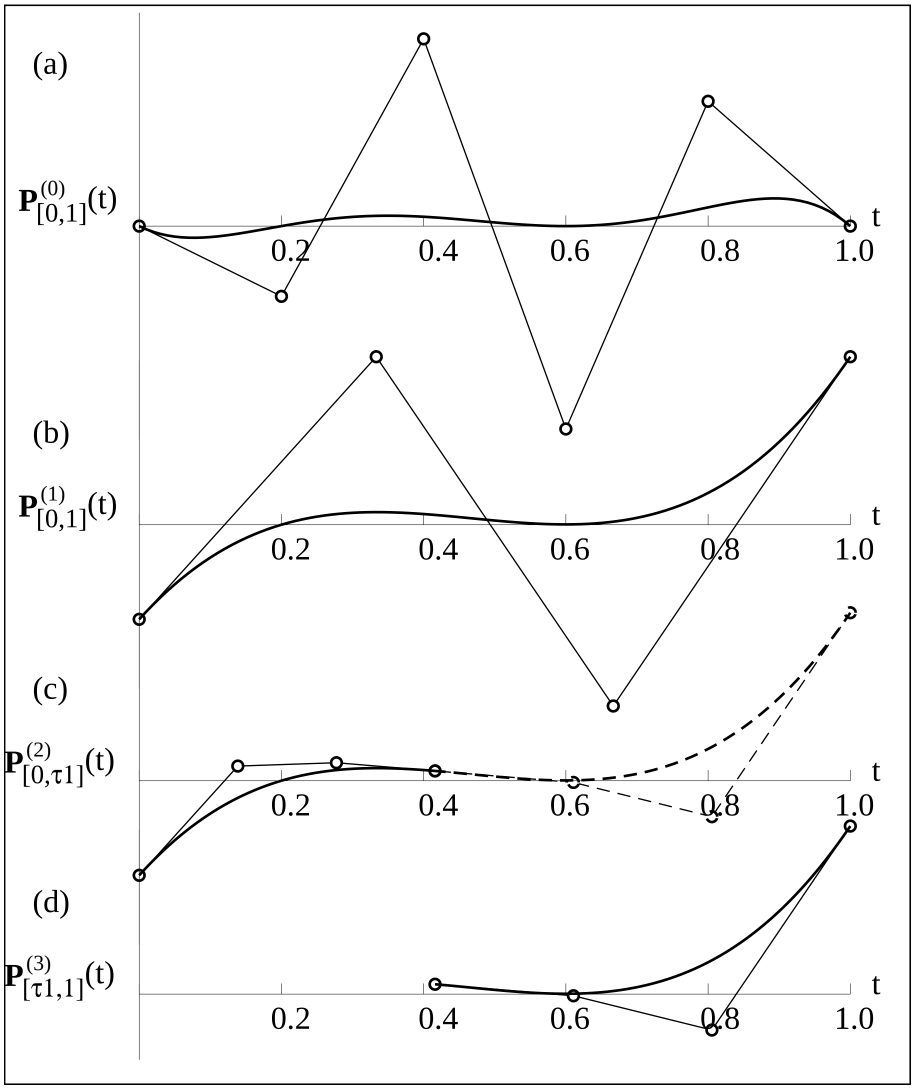
(a)
$\mathbf{P}^{(0)}_{[0,1]}(t)$
t
0.2
0.4
0.6
0.8
1.0
(b)
$\mathbf{P}^{(1)}_{[0,1]}(t)$
t
0.2
0.4
0.6
0.8
1.0
(c)
$\mathbf{P}^{(2)}_{[0,\tau 1]}(t)$
t
0.2
0.4
0.6
0.8
1.0
(d)
$\mathbf{P}^{(3)}_{[\tau 1,1]}(t)$
t
0.2
0.4
0.6
0.8
1.0

Figure 9.2: Root isolation heuristic (a-d).

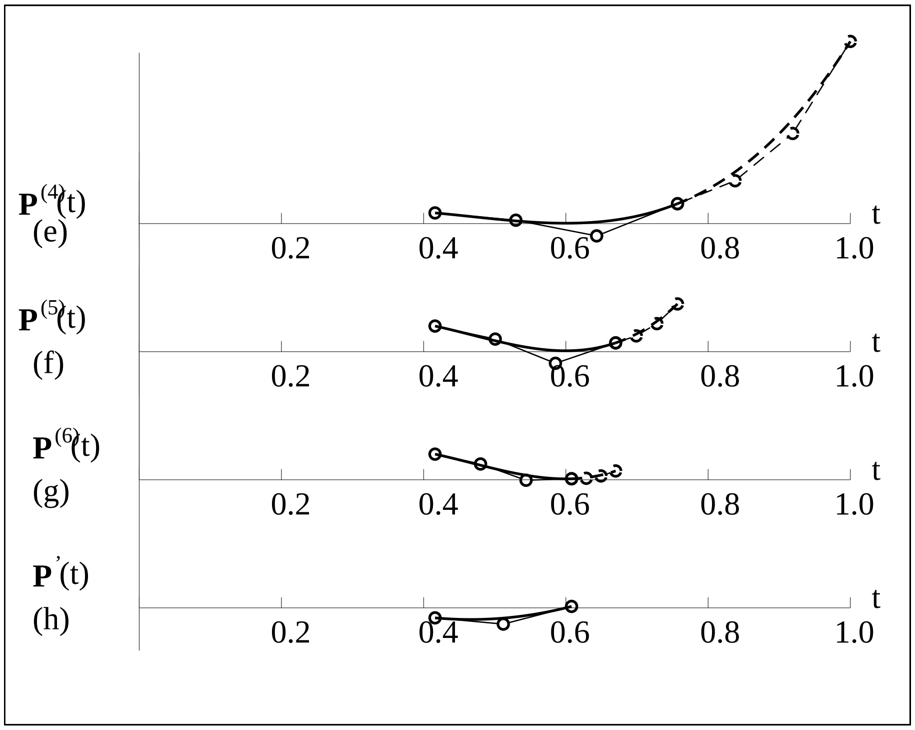

Figure 9.3: Root isolation heuristic (e-h).

9.3 Multiplication of Polynomials in Bernstein Form

Given

$$f(t) = \sum_{i=0}^{m} f_i B_i^m(t), \quad \text{and} \quad g(t) = \sum_{i=0}^{n} g_i B_i^n(t)$$

their product $h(t) = f(t)g(t)$ is a polynomial in Bernstein form of degree $m+n$ whose coefficients are

$$h_k = \frac{\sum_{i+j=k} \binom{m}{i} f_i \binom{n}{j} g_j}{\binom{m+n}{k}}$$

The notation $\sum_{i+j=k}$ means to use all valid values of i and j that add up to k. Denote

$$\langle f_0, f_1, \ldots, f_m \rangle^m(t) = \sum_{i=0}^{m} f_i B_i^m(t).$$

Then, for example,

$$\langle f_0, f_1, f_2 \rangle^2(t) \times \langle g_0, g_1, g_2, g_3 \rangle^3(t) =$$
$$\langle f_0 g_0, \frac{3f_0g_1 + 2f_1g_0}{5}, \frac{3f_0g_2 + 6f_1g_1 + f_2g_0}{10}, \frac{f_0g_3 + 6f_1g_2 + 3f_2g_1}{10}, \frac{2f_1g_3 + 3f_2g_2}{5}, f_2g_3 \rangle$$

The binomial coefficients $\binom{n}{i}$ can be computed using the recurrence relation

$$\binom{n}{i} = \frac{n-i+1}{i} \binom{n}{i-1}.$$

Thus, $\binom{n}{0} = 1$, $\binom{n}{1} = n$, $\binom{n}{2} = \frac{n-1}{2}\binom{n}{1}$, etc.

Bernstein-basis polynomials can be defined over any domain $[t_0, t_1]$, and the equation is given in Equation 2.3 from Section 2.2 (repeated here for convenience):

$$\mathbf{P}_{[t_0,t_1]}(t) = \frac{\sum_{i=0}^{n} \binom{n}{i} (t_1 - t)^{n-i} (t - t_0)^i \mathbf{P}_i}{(t_1 - t_0)^n} = \sum_{i=0}^{n} \binom{n}{i} \left(\frac{t_1 - t}{t_1 - t_0}\right)^{n-i} \left(\frac{t - t_0}{t_1 - t_0}\right)^i \mathbf{P_i}.$$

The algorithm just described for multiplying two Bernstein-basis polynomials over the $[0, 1]$ domain applies to an arbitrary domain $[t_0, t_1]$ without modification.

9.4 Creating a Bernstein-Basis Polynomial with Specified Roots

The multiplication algorithm enables us to create Bernstein-basis polynomials over an arbitrary domain $[t_0, t_1]$ that have specified roots, r_i, $i = 1, \ldots, n$. This is accomplished by multiplying the degree-one polynomials

$$p_{[t_0,t_1]}(t) = \langle x_0 - r_1, x_n - r_1 \rangle_{[t_0,t_1]} * \langle x_0 - r_2, x_n - r_2 \rangle_{[t_0,t_1]} * \cdots * \langle x_0 - r_n, x_n - r_n \rangle_{[t_0,t_1]}.$$

A Bernstein-basis polynomial $\mathbf{P}_{[t_0,t_1]}(t)$ can be represented as an explicit Bézier curve by setting the x-coordinates of the control points equal to $x_i = t_0 + \frac{i}{n}(t_1 - t_0)$.

9.5 Intersection between a Line and a Rational Bézier Curve

The points of intersection between a rational Bézier curve

$$\mathbf{P}(t) = \frac{\sum_{i=0}^{n} w_i \mathbf{P}_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)}$$

and a line

$$ax + by + cw = 0$$

can be computed by substituting the parametric equation of the curve into the implicit equation of the line, producing a polynomial in Bernstein form

$$f(t) = \sum_{i=0}^{n} f_i B_i^n(t), \qquad f_i = ax_i w_i + by_i w_i + cw_i.$$

The roots of $f(t)$ give the parameter values of the points at which the curve intersects the line.