Polynomial Real Root Finding in Bernstein Form

A Dissertation

Presented to the

Department of Civil Engineering

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Melvin R. Spencer

August 1994

This dissertation by Melvin R. Spencer is accepted in its present form by the Department of Civil Engineering of Brigham Young University as satisfying the disseration requirement for the degree of Doctor of Philosophy.

Thomas W. Sederberg, Committee Chairman

Norman L. Jones, Committee Member

Rida T. Farouki, Committee Member

Date

S. Olani Durrant Department Chairman

ACKNOWLEDGEMENTS

I am deeply grateful and would like to thank Dr. Thomas W. Sederberg for this disseration, being my advisory chair, research assistantships, and the lucid insight he provided throughout my graduate work. Without his constant supportive source of ideas, suggestions, encouragement, motivation, and dedication this work would not have met fruition. I also thank him for his patient, considerate demeanor and grace under pressure.

I am very indebted and would like to thank Dr. Henry N. Christiansen for permitting an enthusiastic under-graduate to pursue knowledge and gain enlightenment through the many opportunities provided and with the many talented colleagues associated through his graduate program at Brigham Young University.

I thank the faculty, deans, and administrators of the Department of Civil Engineering, the College of Engineering and Technology, and the Office of Graduate Studies of Brigham Young University for their continual support in granting the many considerations requested for the fulfillment of this degree. In addtion, thanks to the staff at the University Library and its Inter-Library Loan Department for their constant service and unending supply of reference material.

I extend my sincere appreciation and thanks to Dr. Thomas W. Sederberg and Dr. Rida T. Farouki for their significant and incessant contributions towards the completion of this document which include their derivation and presentation of the chapter on preconditioning which is the core of this work, their insights regarding Bernstein form polynomial real root finding, the many days sacrificed and dedicated to the composition, rewriting, and editing, and above all their considerate, good-natured personalities and much appreciated and welcomed suggestions which lessened the burden of the oral and written presentation of this thesis.

Special effort needs to be recognized and thanks extended to my advisory committee Dr. Thomas W. Sederberg, Dr. Norman L. Jones, and Dr. Rida T. Farouke as well as my oral committee chairman Dr. Henry N. Christiansen, and members Dr. Thomas W. Sederberg, Dr. Norman L. Jones, Dr. Rida T. Farouki, and Dr. C. Gregory Jensen for their preparation, examination, and suggestions regarding the content of this dissertation.

I would like recognize and thank my fellow graduate colleagues for their assistance in preparing this work. Dr. Alan K. Zundel for his many contributions to this work which include his insights into the multiple root-finding strategy and its implementation and testing in his algebraic surface rendering program; and particularly his last minute efforts and contributions to the performance assessment and testing of the various algorithms. Kris Klimaszewski for his initial implementation of the degree six isolator polynomial algorithm as well as his many programming insights and comradery. Peisheng Gao and Hong Mu for their unending and expert assistance in programming, illustrations, and testing which they performed simultaneously with their many other graduate duties and responsibilities.

I would like to thank Dr. Anders N. Grimsrud and my other associates at work, for the additional responsibility they have shouldered by providing me a leave of absence from work to finish this academic goal and project.

I would like to thank my parents, relatives, and friends for their emotional support and confidence regarding the completion of this work. Especially my father, Melvin Reed Spencer; and my brother and his family Robert and Shannon, Erin, and Chase Spencer for their financial and emotional support, providing a weekend refuge to refresh and collect my thoughts.

And last but sincerely not least, I am grateful and thankful for the divine inspiration empowered to all concerned for the motivation and exchange supporting the conception, development, and completion of this collaborative effort.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation addresses the problem of approximating, in floating–point arithmetic, all real roots (simple, clustered, and multiple) over the unit interval of polynomials in Bernstein form with real coefficients.

The Bernstein polynomial basis enjoys widespread use in the fields of computer aided geometric design (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. 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 [SP86]. Other examples from CAGD include finding a point on each loop of the intersection curve of two surfaces [Far86], and the "trimming" of offset and bisector curves [FJ94, FN90a]. Examples from computer graphics include ray tracing [Kaj82, Han83, SA84, Wij84, BDM84] and algebraic surface rendering [Sed85, Zun89].

In the early years of computer graphics and CAGD, the tendency was to convert from

the Bernstein to the power basis for root finding (see, for example, [Kaj82]) 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.

The research reported in this dissertation falls in the domain of "experimental computer science and engineering". Many of the results are empirical. Fortunately, there is now growing recognition that proof of performance through such experimental means is an acceptable and even crucial form of computer research [ECS94].

Nearly 25 years ago, Henrici remarked, at a symposium on the numerical solutions of nonlinear problems:

> *I do not feel that the polynomial problem has been solved perfectly, not even from a practical point of view* [Hen70].

Our study of root finding for polynomials in Bernstein form suggests that the field remains fruitful even now.

1.1 Literature Search

The first known work on our problem is an algorithm, due to Lane and Riesenfeld [LR81], which repeatedly applies the de Casteljau algorithm to isolate and refine roots of polynomials in Bernstein form.

Several of the algorithms presented in this dissertation were first documented in an unpublished report drafted nearly ten years ago [SSd86]. The significant developments on the numerical stability of the Bernstein form that arose in the intervening decade justify our procrastination in submitting that report for publication. We always suspected there was much more waiting to be reported!

[Gra89] presents an algorithm for finding roots of B-spline functions, which is similar to the convex hull root finding algorithm in [SSd86]. [Sch90a] describes basically the same algorithm. An intriguing algorithm based on "parabolic hulls" is introduced in [RKF88]. The approach involves a more sophisticated method for isolating roots. [Roc90] presents a heuristic for accelerating the root isolation process. Each of these algorithms is reviewed more thoroughly in [§ 2.6] in Chapter 2.

This dissertation borrows concepts from the following four fields:

1. Computer Aided Geometric Design and Graphics [SP86, FR87, RKF88, FR88]

2. Numerical Analysis [Act70, Buc66, BFR81, Bor85, Bre73, Dod78, DM72, FMM77, IK66, Ham71, Hen64, Hen82, Hil74, Hou53, Hou70, Mac63, Mar66, McN73, Mer06, Mor83, Non84, Ost60, Pen70, Pet89, PC86, Piz75, PW83, PFTV90, RR78, Ric83, Sta70, Tra64]

3. Theory of Equations [Bor50, BP60, Caj04, Dic22, Mac54, Tur52, Usp48]

4. Higher Algebra [Boc64, Dav27, FS65, Kur80, MP65, Sal85]

1.2 Overview

Chapter 2 outlines some basic concepts pertaining to root finding for polynomials in power and Bernstein form. Chapter 3 discusses numerical considerations for implementing root finding schemes in floating–point arithmetic. Chapter 4 presents one of the most significant results of our research, the fact that root condition for polynomials in Bernstein form can often be significantly improved by shrinking the domain in the earliest stages of coefficient construction. For example, in the curve intersection algorithm mentioned earlier, if one of the curves is split in half before proceeding with the intersection algorithm, the accuracy of the final polynomial roots (which indicate the curve parameter values at the intersection points) can be greatly improved.

Chapter 6 outlines several different algorithms for root finding of polynomials in Bernstein form. Chapter 5 describes basic functions common to those root finding algorithms. Chapter 7 compares the execution speed and numerical precision of the algorithms in chapter 6 and of some general purpose power form polynomial root finding schemes. Chapter 8 concludes and summarizes this work.

Chapter 2

Review of Polynomial Real Root Finding

This chapter reviews basic concepts and algorithms concerned with computing the real roots of polynomials represented in the power and Bernstein bases.

2.1 Polynomial Representations and Properties

We begin by recalling some definitions and properties of polynomials represented in both power and Bernstein form.

2.1.1 Power Form Polynomials

A degree n polynomial in the power basis is defined by

$$f(t) = \sum_{i=0}^{n} p_i t^i \tag{2.1.1.1}$$

with $p_n \neq 0$.

2.1.2 Bernstein Form Polynomials

A degree n polynomial in Bernstein form is given by

$$y_{[0,1]}(t) = \sum_{k=0}^{n} y_k \binom{n}{k} (1-t)^{n-k} t^k \tag{2.1.2.2}$$

where $\binom{n}{k}(1-t)^{n-k}t^k$ is the k^{th} Bernstein basis function of degree n, and y_k are the Bernstein coefficients. The subscript $[0,1]$ indicates that the domain of interest is $t \in [0,1]$.

A polynomial in Bernstein form can be defined over arbitrary domain intervals by introducing the change of variables

$$u = \frac{t-a}{b-a} \tag{2.1.2.3}$$

that maps $t \in [a,b]$ to $u \in [0,1]$. We then have

$$y_{[a,b]}(t) = \sum_{k=0}^{n} y_k \binom{n}{k} (1-u)^{n-k} u^k = \sum_{k=0}^{n} y_k \binom{n}{k} \frac{(b-t)^{n-k}(t-a)^k}{(b-a)^n} \tag{2.1.2.4}$$

Explicit Bézier Curves

Useful geometric properties may be associated with a polynomial in Bernstein form by casting it as an explicit Bézier curve (see Figure 2.1):

$$\mathbf{P}_{[a,b]}(t) = (t, y_{[a,b]}(t)) = \sum_{k=0}^{n} \mathbf{P}_k \binom{n}{k} (1-u)^{n-k} u^k \tag{2.1.2.5}$$

where the *control points* $\mathbf{P}_k$ are evenly spaced along the horizontal axis:

$$\mathbf{P}_k = (a + \frac{k}{n}(b-a), y_k). \tag{2.1.2.6}$$

The *control polygon* is formed by the set of line segments connecting adjacent control points. The phrase *control polygon* is not an entirely accurate label, since it is not actually a closed polygon because $\mathbf{P}_n$ and $\mathbf{P}_0$ are not connected.

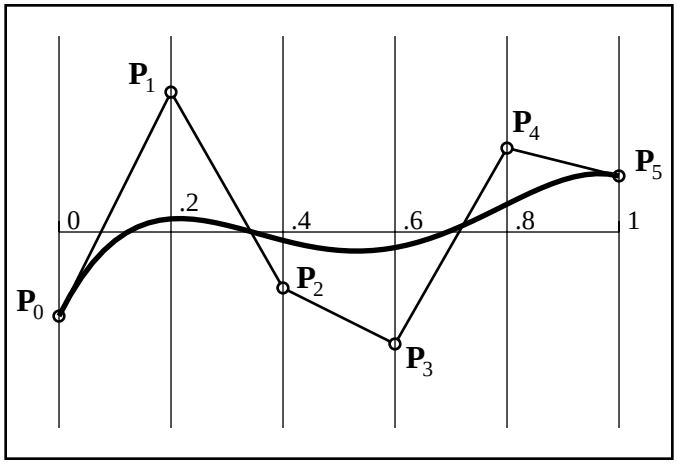

Figure 2.1: Bernstein polynomial as an explicit Bézier curve

Convex Hull Property

We will make extensive use of the convex hull property of Bézier curves, which states that a Bézier curve lies entirely within the convex hull of its control points, as illustrated in Figure 2.2. The convex hull property is assured because the Bernstein polynomial basis functions $\binom{n}{k}\frac{(b-t)^{n-k}(t-a)^k}{(b-a)^n}$ sum to one and are non-negative for $t \in [a,b]$.

Variation Diminishing Property

The variation diminishing property is really an expression of Descartes Law of Signs in the Bernstein polynomial basis. It states that no line intersects a Bézier curve more times than it intersects the control polygon. More specifically, if a given line intersects a Bézier curve n_1 times and the same line intersects the control polygon n_2 times, then

$$n_1 = n_2 - 2n_3 \tag{2.1.2.7}$$

where n_3 is a non-negative integer. For example, for the case of the t–axis in Figure 2.1, $n_1 = n_2 = 3$. Equation (2.1.2.7) requires us to count multiple intersections properly, so a simple tangent intersection accounts for two intersections.

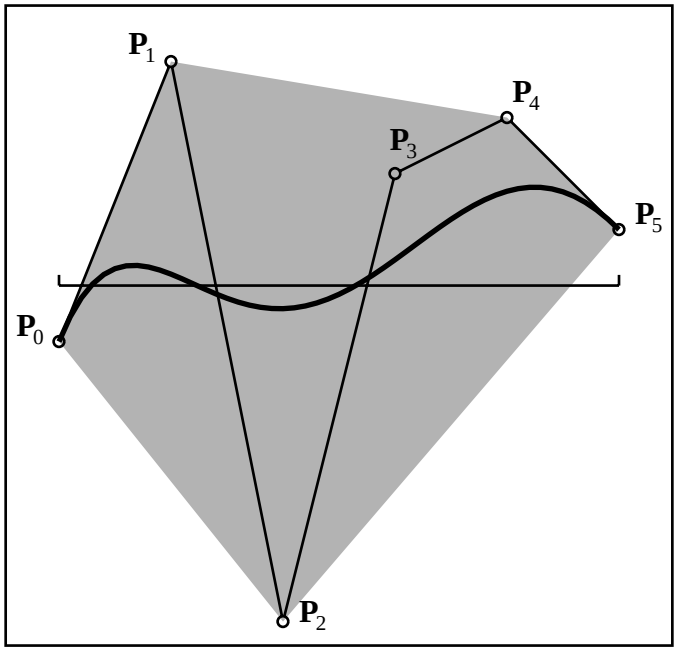

Figure 2.2: Convex hull property

Two implications of the variation diminishing property are immediately apparent. First, if all coefficients of $y_{[a,b]}(t)$ have the same sign, there can be no roots in $[a,b]$. Second, if the control polygon of $\mathbf{P}_{[a,b]}(t)$ crosses the t–axis exactly once, then $y_{[a,b]}(t)$ has exactly one root in $[a,b]$.

2.2 Polynomial Real Root Localization

The first step for many root finding methods is to divide the domain into disjoint intervals which each contain zero or one root, a procedure known as *root isolation*. The more generic term *localization* [Hou70] refers to root isolation as well as the determination of global root bounds. Localization schemes are usually based on examining simple expressions involving only the polynomial coefficients.

This section is limited to a discussion of real root localization methods. Complex root localization techniques are outlined in [Hou70, Mar66]. If a polynomial has real coefficients,

then any complex roots must occur in conjugate pairs. The following three subsections review techniques for globally bounding, and locally isolating, distinct as well as multiple real roots.

2.2.1 Techniques for Bounding Real Roots

This section reviews techniques for determining global bounds for real roots of polynomials in power form. Concepts relevant to bounding real roots of polynomials in Bernstein form are presented in [§ 5.7].

The real roots of polynomials expressed in power form can lie anywhere in the open interval $(-\infty, +\infty)$. The only way a degree n polynomial in power form can have a root exactly at infinity is if its leading coefficient is $p_n = 0$.

Here we review four of several classical methods for determining a tighter upper bound on the positive real roots of a power basis polynomial [Kur80, MP65, BP60, Caj04]. With some simple transformations, it is possible to use these bounding methods to also find lower bounds on positive roots, as well as upper and lower bounds on negative roots.

Suppose a method exists which guarantees that all real roots of $f(t)$ are less than an upper bound B_1. To compute a lower bound for the positive real roots of $f(t)$, find the upper bound B_2 of the roots of $\Phi_2 \equiv t^n f(1/t)$. Then, a lower bound for the positive roots of $f(t)$ is $1/B_2$ since if ρ is a positive root of $f(t)$, then $1/\rho$ will be a root of $\Phi_2(t)$ and if $1/\rho < B_2$, then $\rho > 1/B_2$. Likewise, if B_3 is an upper bound on the roots of $\Phi_3(t) \equiv f(-t)$, then $-B_3$ is a lower bound on the negative roots of $f(t)$; and if B_4 is an upper bound on the roots of $\Phi_4 \equiv t^n f(-1/t)$, then $-1/B_4$ is an upper bound on the negative roots of $f(t)$.

Listed below are several of the more convenient and less computationally–intensive methods for finding positive upper bounds on the real roots of a polynomial. Others not cited can be found in [Hou70, Mar66, FS65]. Many of these theorems are simplified versions of their more general forms attributed to bounding complex roots of a polynomial found in

Table 2.1: Global Bounding of Power Form Polynomial Real Roots

Positive roots of $f(t)$ lie in $[1/B_2, B_1]$

Negative roots of $f(t)$ lie in $[-B_3, -1/B_4]$

Polynomial	*Coefficients*	*Upper bound of roots*
$\Phi_1(t) \equiv f(t)$	$f_0, f_1, \ldots, f_n$	B_1
$\Phi_2(t) \equiv t^n f(1/t)$	$f_n, f_{n-1}, \ldots, f_0$	B_2
$\Phi_3(t) \equiv f(-t)$	$f_0, -f_1, f_2, \ldots, (-1)^n f_n$	B_3
$\Phi_4(t) \equiv t^n f(-t)$	$(-1)^n f_n, (-1)^{n-1} f_{n-1}, \ldots, f_0$	B_4

[Hou70, Mar66]. In the following discussion, we assume that $f(t) = \sum a_i t^i$ is a degree n monic polynomial ($a_n = 1$).

The *Cauchy bound* [Kur80, MP65, Hou70] is usually too conservative when one is only interested in real roots [Kur80], although it does provide a basis for other methods.

Theorem 2.2.1.1 (Cauchy Bound) *All real roots of $f(t) \in \Re[t]$ are contained in the closed interval $[-B, B]$, where*

$$B = 1 + \max_i(|a_i|). \tag{2.2.1.8}$$

The *modified Cauchy bound* [Mac54, Dav27, Mer06, Caj04] has the Cauchy bound for its worst case, and thus may give a closer bound.

Theorem 2.2.1.2 (Modified Cauchy Bound) *Let N be the absolute value of the most negative coefficient of $f(t) \in \Re[t]$. Then*

$$B = 1 + N \tag{2.2.1.9}$$

is a positive upper bound for all real roots of $f(t)$.

The *negative inverse sum bound* [BP60, Dic22, Caj04] and the *Maclaurin bound* [Kur80, MP65, FS65, BP60, Dic22, Mer06] are nearly always much better, although still rather conservative. Examples in [BP60, Dic22] illustrate that both of these bounds are polynomial dependent; either one may yield a better bound than the other.

Theorem 2.2.1.3 (Negative Inverse Sum Bound) *For each negative coefficient a_j of $f(t) \in \Re[t]$, let S_j be the sum of all positive coefficients following a_j in the sequence*

$a_0, \ldots, a_n$. *Then*

$$B = 1 + \max_j \left(\left| \frac{a_j}{S_j} \right| \right) \tag{2.2.1.10}$$

is a positive upper bound for all real roots of $f(t)$.

Theorem 2.2.1.4 (Maclaurin Bound) *Let* N *be the absolute value of the most negative coefficient of* $f(t) \in \Re[t]$, *and let* k *be the index of the last negative coefficient in the sequence* $a_0, \ldots, a_n$. *Then*

$$B = 1 + N^{1/(n-k)} \tag{2.2.1.11}$$

is a positive upper bound for all real roots of $f(t)$.

The *term grouping bound* [MP65, BP60] and the *Newton bound* [Kur80, MP65, BP60, Mac54, Tur52] both require initial guesses, and are more computationally intensive, but they ordinarily provide closer bounds than any of the preceding methods [BP60], with the Newton bound providing the tightest bound on real roots.

Theorem 2.2.1.5 (Term Grouping Bound) *Let* $f(t) \in \Re[t]$ *be expressed in the form* $f_1(t) + f_2(t) + \ldots + f_k(t)$, *such that the ordered coefficients of each of the polynomials* $f_i(t)$ *exhibit only one change of sign, the coefficients of the highest–degree terms being positive. Then any positive number* B *that renders* $f_i(B) > 0$ *for* $i = 1, \ldots, k$ *is an upper bound for all real roots of* $f(t)$.

Theorem 2.2.1.6 (Newton Bound) *Let* $f(t) \in \Re[t]$ *and let* B *be a number such that* $f(B)$, $f'(B)$, $f''(B)$, $\ldots$, $f^{(n)}(B)$ *are all positive. Then* B *is a positive upper bound for all real roots of* $f(t)$.

The values for $f(B)$, $f'(B)$, $\ldots$, $f^n(B)$ are easily found using Horner's algorithm [§ 5.2.1], [MP65, Mac54].

2.2.2 Isolation Techniques for Distinct Real Roots

The classical literature is abundant with root isolation methods that separate the distinct real roots of a polynomial. Root isolation is an important step in solving for the roots of a polynomial, since many iterative real root approximations methods require a "sufficiently

close" starting value or a narrow inclusion interval to guarantee that their sequences of approximations converge efficiently to a root. We now review some of the salient concepts for isolating real roots.

Polynomial Factors and Roots

The result of substituting a number α for x into a polynomial $f(x)$ is a number called the *value* $f(\alpha)$ of the polynomial at $x = \alpha$ [Usp48]. If $f(\alpha)$ is equal to zero the polynomial f is said to vanish, and α is called a *zero* or *root* of polynomial f [Bor50].

An *algebraic equation* of degree n is an expression formed by equating a non–zero degree–n polynomial $f(x) \in \mathcal{F}[x]$ to zero [Caj04, Usp48, Bor50, Tur52],

$$f(x) = 0. \tag{2.2.2.12}$$

Let $f(x) \in \mathcal{F}[x]$ and suppose that $\mathcal{K}$ is a field containing $\mathcal{F}$, i.e., $\mathcal{K}$ is an extension of $\mathcal{F}$ such that $\mathcal{F} \subset \mathcal{K}$ [Bor50]. The result of substituting any number $\alpha \in \mathcal{K}$ for x into the polynomial $f(x)$ yields

$$f(\alpha) = \sum_{i=0}^{n} a_i \alpha^i = \beta,$$

where $\beta \in \mathcal{K}$ is the *value* of f at $x = \alpha$. By definition, α is called a *zero*, *root*, or *solution* of the equation $f(x) = 0$ when $\beta = 0$ [BP60, Hou70].

The existence of a root of any algebraic equation, or non-zero polynomial, is guaranteed by the *fundamental theorem of algebra* [Dic22, Boc64, MP65, Ric83].

Theorem 2.2.2.1 (The Fundamental Theorem of Algebra) *Any non–zero polynomial of degree n ($a_n \neq 0$) in $\mathcal{C}[x]$ (and thus in $\mathcal{R}[x]$) always has at least one complex* [1] *(real or imaginary) root.*

In other words, *every algebraic equation has at least one solution* [Caj04, Usp48, Tur52,

[1] A *complex number* $a + ib$, with $a, b \in \mathcal{R}$, $i = \sqrt{-1}$, is *real* if $b = 0$, *imaginary* (or *nonreal*) if $b \neq 0$, and *pure imaginary* if $a = 0$.

Hou70, Hil74]. This theorem only guarantees the existence of a root; determining the number and type of the roots of $f(x)$ is more involved.

The definitions of *value* and *root* of a polynomial may also be established on the basis of the *remainder theorem* [Caj04, Dic22, Usp48, Bor50, Mac54, Hou70, Kur80], and its corollary the *factor theorem* [Caj04, Dic22, Bor50, Mac54, Hou70, Kur80], both a consequence of the *polynomial division algorithm* represented by equation (2.2.2.13) and the conditions (2.2.2.14) and (2.2.2.15).

Theorem 2.2.2.2 (Polynomial Division Algorithm) *For any two polynomials $f(x)$, $f_1(x) \in \mathcal{F}[x]$ (called the* dividend *and the* divisor *polynomials, respectively), where $f_1(x) \neq 0$, there exists an unique pair of polynomials $q(x), r(x) \in \mathcal{F}[x]$ (the* quotient *and* remainder *polynomials) such that*

$$f(x) \equiv f_1(x)q(x) + r(x) \tag{2.2.2.13}$$

with either

$$Deg(r) < Deg(f_1) \tag{2.2.2.14}$$

or

$$r(x) = 0. \tag{2.2.2.15}$$

Theorem 2.2.2.3 (The Remainder Theorem) *If the polynomial $f(x)$ is divided by the linear term $x - \alpha$, where α is any number, then the remainder is a constant, equal to the value $f(\alpha)$ of $f(x)$ at $x = \alpha$, i.e.,*

$$f(x) \equiv (x - \alpha)q(x) + f(\alpha). \qquad (Deg(f) = n) \tag{2.2.2.16}$$

Theorem 2.2.2.4 (The Factor Theorem) *The remainder $f(\alpha)$ vanishes iff $x - \alpha$ is a linear factor of the polynomial $f(x)$. Equivalently, α is a root of $f(x) = 0$ iff $x - \alpha$ divides exactly (i.e., with zero remainder) into $f(x)$, so that*

$$f(x) \equiv (x - \alpha)q(x). \qquad (Deg(f) = n,\ Deg(q) = n - 1) \tag{2.2.2.17}$$

Together the fundamental theorem and the factor theorem indicate the number of linear factors, and hence the number of roots, of a polynomial degree n. These results are succinctly described by the following three theorems [Caj04, Bor50, Dic22, Ham71, Hen64, Hou53, Hou70, Kur80, Usp48].

Theorem 2.2.2.5 (Factored, or Product, Form Theorem) *If $Deg(f) = n \geq 1$, then $f(x) \in \mathcal{C}[x]$ is uniquely expressible in* factored form *as the product of n monic linear factors $x - \alpha_i$ and the constant a_n, where $\alpha_1, \ldots, \alpha_n$ denote the roots corresponding to the linear factors:*

$$f(x) \equiv a_n(x - \alpha_1)(x - \alpha_2)\cdots(x - \alpha_n) \quad (a_n \neq 0). \tag{2.2.2.18}$$

Theorem 2.2.2.6 (Repeated Factored Form Theorem) *If $Deg(f) = n \geq 1$, then $f(x) \in \mathcal{C}[x]$ is uniquely expressible in* repeated factored form *as the product of $k \leq n$ distinct monic power factors $(x - \alpha_j)^{m_j}$ and the constant a_n, where the positive integers m_j sum to n and denote the multiplicities of the distinct roots $\alpha_1, \ldots, \alpha_k$, i.e.,*

$$f(x) \equiv a_n(x - \alpha_1)^{m_1}(x - \alpha_2)^{m_2}\cdots(x - \alpha_k)^{m_k} \quad (a_n \neq 0,\ n = \sum_{j=1}^{k} m_j). \tag{2.2.2.19}$$

Theorem 2.2.2.7 (Root Census Theorem) *An algebraic equation of degree n may be regarded as having exactly n roots (not necessarily distinct), if the roots are counted according to their multiplicities.*

Each factor $(x - \alpha_j)^{m_j}$ corresponds to an *m_j-fold or m_j-tuple* root or a *multiple* root of multiplicity m_j [Bor50, MP65]. When $m_j = 1$, a root α_j is called a *simple* or *distinct* root. Roots of multiplicity $m_j = 2, 3, 4, \ldots$ are referred to as *double, triple, quadruple, ...* roots. From a numerical perspective, groups of roots that are nearly multiple form a "cluster" of roots and are termed *root clusters* [Hou70]. Multiple roots are necessarily simultaneous roots of $f(x)$ and successive derivatives of $f(x)$ [Usp48], i.e.,

$$\begin{aligned} f(\alpha) = 0, &\qquad f'(\alpha) \neq 0 \\ f(\alpha) = f'(\alpha) = 0, &\qquad f''(\alpha) \neq 0 \\ f(\alpha) = f'(\alpha) = f''(\alpha) = 0, &\qquad f'''(\alpha) \neq 0 \\ \ldots &\qquad \ldots, \end{aligned}$$

for a simple, double, triple, ...root α.

The first derivative of the polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$ can be represented in the repeated factored form of equation (2.2.2.19) as

$$f'(x) = \sum_{i=0}^{n-1}(i+1)a_{i+1}x^i = \sum_{j=1}^{k} \frac{m_j\ f(x)}{(x - \alpha_j)} \tag{2.2.2.20}$$

and has as its first derivative $f''(x)$, and so on [Caj04, Dic22, Kur80, Mac54].

For any polynomial in $\mathcal{C}[x]$, the number of roots is synonymous with the number of its linear factors. A discussion of polynomials in $\mathcal{R}[x]$ requires two additional theorems on *conjugate pair roots* and *real factored forms* [Bor50, Kur80, Mac54, McN73, Usp48].

Theorem 2.2.2.8 (Conjugate Pair Roots Theorem) *If a polynomial in $\mathcal{R}[x]$ has an imaginary root $\beta = a + ib$ of multiplicity m, it has also the conjugate value $\overline{\beta} = a - ib$ as a root of the same multiplicity, i.e., imaginary roots occur in conjugate pairs.*

Theorem 2.2.2.9 (Real Factored Form Theorem) *Every polynomial $f(x)$ of degree $n > 2$ in $\mathcal{R}[x]$ can be expressed as a product of linear and/or quadratic factors that are irreducible monic polynomials in $\mathcal{R}[x]$, that is*

$$f(x) \equiv a_n(x-\alpha_1)^{m_1}\cdots(x-\alpha_i)^{m_i}(x^2+b_1x+c_1)^{l_1}\cdots(x^2+b_jx+c_j)^{l_j}, \qquad (2.2.2.21)$$

where

$$x^2 + b_jx + c_j = (x-\beta_j)(x-\overline{\beta_j}). \qquad (2.2.2.22)$$

Location Principles

Rolle's Theorem [Tur52] relates the locations of the roots of $f(x)$ and the roots of its derivatives.

Theorem 2.2.2.10 (Rolle's Theorem) *Between any two consecutive real roots a and b of the polynomial $f(x)$ there lies an odd number of roots — and therefore at least one root — of its derivative polynomial $f'(x)$, a root of multiplicity m being counted as m roots.*

Rolle's theorem provides the following concepts for isolating real roots of a polynomial $f(x)$ based on the real roots of its derivative polynomial $f'(x)$.

1. Between any two consecutive real roots c and d of $f'(x)$ there lies at most one real root of $f(x)$.
2. The smallest and largest real roots, α and ω, of $f'(x)$ constrain the smallest and largest real roots of $f(x)$ to lie in the intervals $[-\infty, \alpha]$ and $[\omega, +\infty]$, respectively.
3. Multiple roots of $f(x)$ are necessarily roots of $f'(x)$.

4. A method for separating the real roots of a polynomial $f(x)$ whose roots are all simple, when all the real roots of $f'(x)$ are known.

Variation of Coefficient Signs

The number of sign variations in the sequence of coefficients $a_0, a_1, \ldots, a_n$ of a degree–n polynomial $f(x)$ is equal to the number of consecutive pairs of coefficients that are of opposite signs, after zero values have been deleted from the sequence.

Theorem 2.2.2.11 (Descartes' Rule of Signs) *A real polynomial f(t) cannot have more positive real roots than there are changes of sign in its sequence of coefficients; nor more negative real roots than there are changes of sign in the sequence of coefficients $f(-t)$.*

Thus, the number of positive or negative roots will be even if there is an even number of variations of sign, and odd if there is an odd number of changes.

Sturm Sequence

Descartes' rule provides only an upper bound on the number of real roots of a polynomial. A method that determines the exact number of distinct roots of a polynomial $f(t)$ on an interval (a, b) is based on computing a *Sturm sequence* $f_1(t), \ldots, f_m(t)$ for $f(t)$, defined by

$$\begin{aligned} f_1(t) &= f(t) \\ f_2(t) &= f'(t) \\ f_i(t) &= q_i(t)\, f_{i+1}(t) - f_{i+2}(t) \quad (i \geq 1). \end{aligned}$$

The sequence terminates upon encountering a polynomial $f_m(t)$, possibly a constant, that does not vanish on the interval of interest.

Theorem 2.2.2.12 (Sturm's Theorem) *Let $f_1(t), f_2(t), \ldots, f_m(t)$ be a Sturm sequence for the real polynomial $f(t)$, and let the interval (a, b) be such that neither a nor b is a root of $f(t)$. Then if $S(a)$ and $S(b)$ are the number of sign changes in the sequences of values $f_1(a), f_2(a), \ldots, f_m(a)$ and $f_1(b), f_2(b), \ldots, f_m(b)$, the number of* distinct *real roots of $f(t)$ between a and b is given exactly by $S(a) - S(b)$.*

Note that Sturm's theorem does *not* count roots on (a, b) according to their multiplicities.

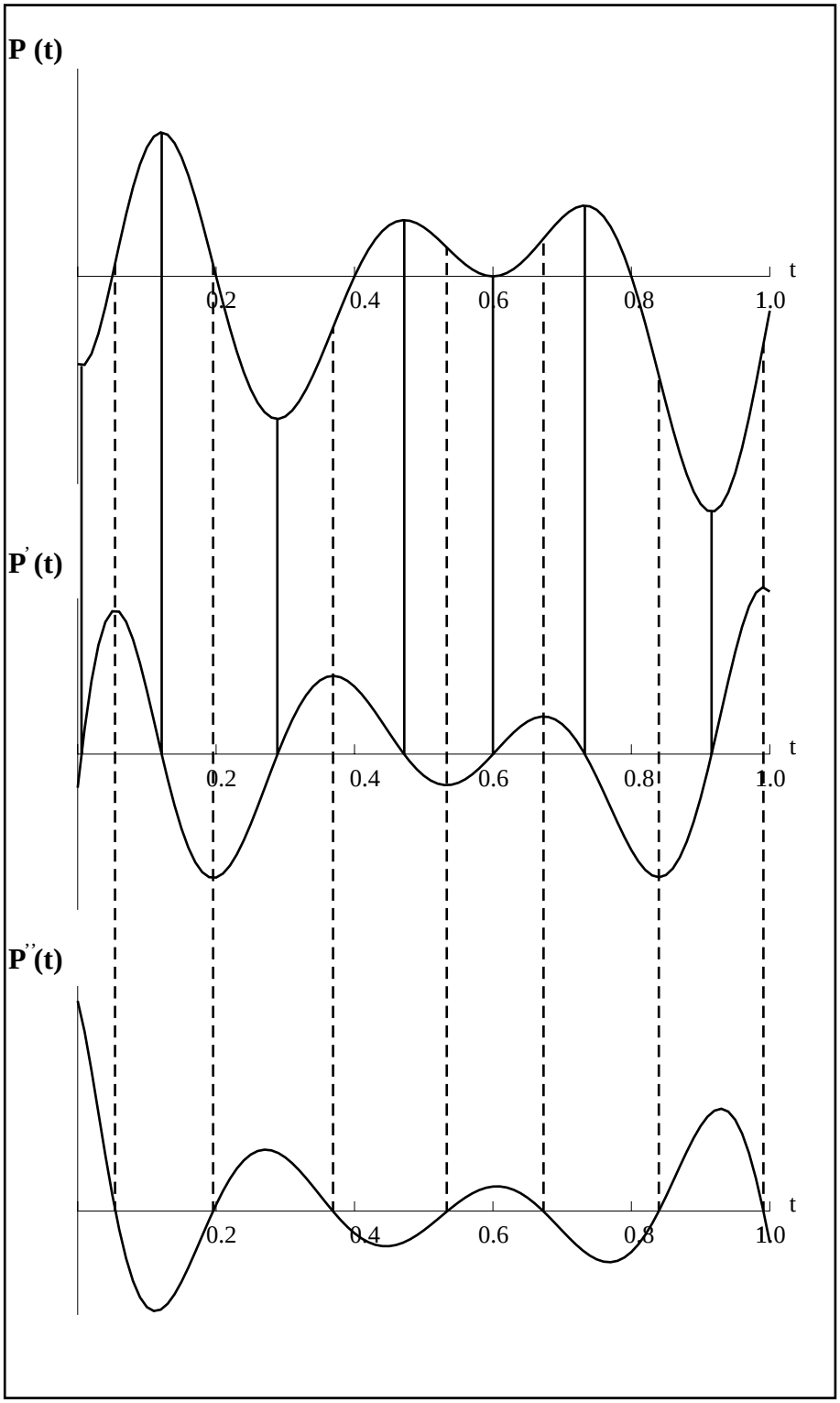

Figure 2.3: Application of Rolle's theorem.

Budan–Fourier Sequence

A weaker result, which relies only on the polynomial and its derivatives rather than a division sequence, is the Budan–Fourier theorem:

Theorem 2.2.2.13 (Budan–Fourier) *Let the interval (a,b) be such that neither a nor b is a root of the polynomial $f(t)$ of degree n. Then if $C(a)$ and $C(b)$ denote the number of sign changes in the sequences of values $f(a), f'(a), \ldots, f^{(n)}(a)$ and $f(b), f'(b), \ldots, f^{(n)}(b)$, the number of roots between a and b, counted according to multiplicity, differs from $C(a) - C(b)$ by at most a positive even amount.*

Isolating Polynomials

Another insightful approach to separating the real zeros of a given polynomial is to utilize the real roots of *isolating polynomials*. Rolle's theorem may be regarded as a corollary to the following theorem of Borofsky [Bor50]:

Theorem 2.2.2.14 *Suppose $f(x)$, $g(x)$, and $h(x)$ are real polynomials, such that a and b are consecutive real roots of $f(x)$, and $g(a)$ and $g(b)$ have the same signs. Then the polynomial $F(x) \equiv g(x)f'(x) + h(x)f(x)$ has an odd number of roots between a and b if each root is counted according to its multiplicity.*

Uspensky [Usp48] describes a related theorem due to de Gua:

Theorem 2.2.2.15 (de Gua) *Let $f(x)$ be a real polynomial of degree n having distinct positive roots $x_1, \ldots, x_r$ with multiplicities $m_1, \ldots, m_r$. Then if we define*

$$F(x) \;=\; x\, f^{'}(x) \;+\; \alpha\, f(x),$$

where α is an arbitrary non–zero constant, the following are true:

1. *$F(x)$ has at least one root in each of the intervals $(x_1, x_2), \ldots, (x_{r-1}, x_r)$;*
2. *for $m_j > 1$ each root x_j is also a root of $F(x)$ with multiplicity $m_j - 1$; and*
3. *all roots of $F(x)$ are real when $\alpha > 0$.*

Similar considerations apply to the negative roots of $f(x)$. A generalized version of the above theorem is presented by Sederberg and Chang in [SC94]; this will be discussed in detail in [§ 6.5].

Finally, Dedieu and Yakoubsohn [DY93] use an *exclusion function* to separate the real roots of a polynomial in their algorithm that is briefly discussed in [§ 2.5.4].

2.2.3 Isolation Techniques Accounting Multiple Real Roots

Multiple real roots lie, in a sense, at the limit between cases of distinct real roots and complex roots [BP60, Dic22, Mer06]. (Dickson [Dic22, p.29] illustrates this principle for the roots of a real quadratic polynomial with a simple ruler–and–compass construction.)

Common Roots of Two Polynomials

The *Euclidean algorithm* for deriving the *greatest common divisor* (GCD) of two polynomials [Van70] provides for:

1. the determination of the multiple roots of a polynomial by reducing it to a polynomial with the same, but simple, roots; and thus
2. the determination of the number of roots of a polynomial over an interval of the real axis.

The GCD of $f(x)$ and $f'(x)$, denoted by

$$g(x) \;=\; (\; f(x),\; f'(x)\;)\,, \tag{2.2.3.23}$$

can be used to reduce the multiple roots of $f(x)$ to simple ones by performing the division

$$h(x) \;=\; \frac{f(x)}{g(x)}, \qquad (Deg(g(x)) \neq 0). \tag{2.2.3.24}$$

The polynomial $h(x)$ then possesses the same roots as $f(x)$, but without multiplicity.

Separation of Multiple Roots

An extension of the GCD algorithm that separates all the multiple roots of a polynomial into distinct linear factors and simultaneously determines the multiplicity of each root is described in [Dun72, Dun74, MP65, Van70] and implemented by [Dun72] for this purpose. Define a *GCD sequence* by

$$\begin{aligned} g_1(x) &= \left(g_0(x),\ g_0'(x) \right), \qquad g_0(x) = f(x) \\ g_2(x) &= \left(g_1(x),\ g_1'(x) \right) \\ &\cdots \\ g_m(x) &= \left(g_{m-1}(x),\ g_{m-1}'(x) \right), \quad Deg(g_m(x)) = 0 \end{aligned}$$

and set

$$\left\{ h_j(x) = \frac{g_{j-1}(x)}{g_j(x)} \right\}_{j=1}^{m}, \tag{2.2.3.25}$$

and then

$$\left\{ F_k(x) = \frac{h_k(x)}{h_{k+1}(x)} \right\}_{k=1}^{m-1}, \quad F_m(x) = h_m(x). \tag{2.2.3.26}$$

All the roots of the polynomials $F_k(x)$ are then *simple* roots, and correspond to *k–fold* roots of $f(x)$. Note that $f(x)$ can not possess roots of multiplicity greater than m.

2.3 Polynomial Real Root Approximation

This section reviews both closed–form algebraic solutions as well as numerical iterative approximations to polynomial equations expressed in power form. The first subsection [§ 2.3.1] discusses closed–form solutions for low–degree equations, with special attention to methods that are concerned only with real roots. The next three subsections classify iterative numerical schemes under three general groups: basic serial iterative methods [§ 2.3.2], combined or hybrid serial iterative methods [§ 2.3.3], and simultaneous iterative methods [§ 2.3.4]. Specific power form polynomial root finders that find all polynomial roots, as well as power

form and Bernstein polynomial root finders that find only the real roots, are reviewed in the subsequent sections [§ 2.4], [§ 2.5] and [§ 2.6].

2.3.1 Closed Form Solvers

Direct root–finding algorithms for equations that are algebraically solvable by radicals (the roots may be expressed explicitly in terms of the coefficients by a general algebraic formula [Lod90, MP65, Bor50, Caj04]) are called *exact*, or *closed–form* solvers. Any polynomial equation of degree ≤ 4 can be solved in closed form; for higher degrees a general closed form solution is impossible [Kur80, MP65].

A thorough discussion of classical solutions for quadratic, cubic, and quartic equations can be found in elementary texts on the theory of equations [BP60, Mac54, Tur52, Bor50, Usp48, Dic22, Mer06, Caj04], and higher algebra [Kur80, MP65, Dav27]. In principle, these methods yield exact results, although floating–point implementations are, of course, subject round–off error and possible ill–conditioning effects.

Although numerical implementations of direct methods are subject to accuracy loss and require square or cube root extractions, special purpose library functions for solving real quadratic and cubic equations have bested general purpose root finding methods by a factor of 7 [MR75]. Closed form solvers restricted to finding only real roots are of particular interest for CAGD and graphics applications, as discussed in [§ 1]. Hanrahan recorded nearly a 25 percent reduction of image rendering CPU time using exact solutions as compared to general polynomial root finders [Han83].

Numerical algorithms returning non-complex solutions are given in [Sch90b]. Numerical considerations addressing the stable computation of closed form root expressions are presented in a report by Lodha [Lod90], and are addressed for the quadratic and cubic in [PFTV90]. Vignes [Vig78] also outlines a stable cubic real root solver developed by M. La Porte.

Further implementation considerations are deferred to [§ 5.9]. Other references discussing numerical implementation of exact solvers include [HE86, PW83, Ric83, BFR81, McN73, Hen64, Hou70, Pen70, Buc66].

2.3.2 Basic Serial Iterative Methods

Methods referred to by [Pet89] that compute one zero at a time and require successive pre–isolation of the roots and post-deflation to remove the corresponding linear factors, and by Householder [Hou70] as *local (convergent) methods* that are dependent on close initial guesses to roots for convergence will be called *serial iterative methods* here. *Basic serial iterative methods* are presented in a variety of numerical texts that discuss iterative solution of the real roots of an equation. Rice [Ric83] outlines several of these basic iterative methods along with their corresponding iterative steps, which include the following schemes:

Bisection Method

Regual Falsi Method

Modified Regula Falsi Method

Secant Method

Mueller's Method (see [Dun72, Mul56, PFTV90])

Fixed-Point Method

Newton's Method

Higher-Order Newton Method

The bisection and regula falsi methods are frequently referred to as *root trapping* or *bracketing* methods because they require a bounding interval containing the real root in their operation. Bracketing methods always converge, but at a linear rate, or superlinear rate depending on what acceleration techniques are applied.

Basic serial iterative schemes may also be characterized by their use of interpolating polynomials [Hil74, Hou70, Ost60, Tra64]:

- Linear interpolation, e.g., as in the secant method.
- Linear inverse interpolation, e.g., the regula falsi methods.
- Quadratic interpolation, e.g., Muller's method.
- Other interpolation–based schemes, e.g., Newton's method.

Any of these methods can be combined, extended, and applied to general purpose polynomial root finding schemes for polynomials in both power and Bernstein forms.

2.3.3 Hybrid Serial Iterative Methods

Hybrid serial methods combine basic iterative root bracketing techniques (which always converge, but usually at a slow rate) with the faster convergent schemes that occasionally exhibit non-convergence [PW83]. The result is a hybrid approximation strategy that always converges at a faster rate by using a higher order convergence technique while the iterative step remains within the specified interval, and the bracketing technique when the result steps out of the interval.

Hybrid methods found in the literature include a *secant–false position* algorithm [PW83] in PASCAL, a *Newton–bisection* algorithm in both FORTRAN and C [PFTV86, PFTV90], a *Newton–regula falsi* algorithm in C by Blinn [Gla89], and an algorithm in FORTRAN and C [FMM77, PFTV86, PFTV90] by Brent [Bre71] based on previous work by Van Wijngaarden and Dekker which combines root bracketing, bisection, and inverse quadratic interpolation to approximate a real root.

2.3.4 Simultaneous Iterative Methods

Methods referred to by Petkovic [Pet89] that simultaneously determine all zeros of a polynomial, and by Householder [Hou70] as *global (convergent) methods* that are not dependent on close initial guesses to roots for convergence are designated here as *simultaneous iterative methods*. Such methods are not the focus of this study, but some of the more established

algorithms are discussed in [§ 2.4] as a reference to their use as benchmarking algorithms compared against the algorithms presented in Chapter 6. Other simultaneous methods mentioned in the literature include Bernoulli's method [Ait26], Graeffee's method [Gra65], the quotient difference method [Mor83, Hen82, BFR81, McN73, Hou70, Hen64] Petkovic's simultaneous inclusion method [Pet89, Pet81], Gargantini's method [Gar79], etc.

2.4 Power Form General Purpose Root Finders

This section reviews some *general purpose root finding methods* found in the literature that approximate all the roots of polynomials in power form. Root finders discussed below include Jenkins and Traub's RPOLY, Dunaway's COMPOSITE, Madsen and Reid's Newton–based, and Laguerre's methods. Additional general purpose methods not outlined here, but referenced in other works, include the Lehmer–Schurr method [PFTV90, Mor83, Dun72, Act70, Hou70, Hou53], Lin's linear and quadratic methods [Hil74, McN73, Act70, Buc66, Lin43, Lin41], Bairstow's Newton quadratic factor method [PFTV90, PC86, Mor83, Hen82, RR78, Piz75, Hil74, McN73, Dun72, Ham71, Hou70, Buc66, IK66, Hen64] as well as other higher order methods referenced in [HP77, Hil74, Dun72, Hou53].

Jenkins–Traub's Method

Jenkin's RPOLY FORTRAN program, listed in [Jen75], finds all the zeros of real polynomials in power form. RPOLY is based on a three–stage algorithm described in [JT70] which extracts an approximate linear or quadratic factor from a sequence of polynomials of degree one less than the original polynomial that are generated by various shifting operations. The real zero or quadratic factor is then deflated from the original polynomial and the process repeats on the reduced polynomial until all zeros are approximated. The first and second stages are linear convergent, and the third stage is superquadratic convergent and thus usually requires only a few iterations. Termination criteria for stage three iterations

are based on round–off error analysis. Due to its robust and efficient performance, RPOLY has become established as a de facto state–of–the–art algorithm that provides a basis of comparison for many root finding algorithms. Jenkins and Traub's algorithm implemented for real polynomials reported *about* $2n^2$ *milliseconds to calculate all the zeros* for degree $n \geq 20$ real polynomials, in contrast to about $8n^2$ milliseconds for their complex version [JT70]. Other polynomial root finding publications that refer to these methods include [PFTV90, Ric83, BFR81, RR78, MR75, Dun72, Hou70].

Dunaway's COMPOSITE Method

Dunaway's COMPOSITE FORTRAN algorithm is included in [Dun72]. It computes all the roots of real power form polynomials, with particular emphasis on the problem of solving polynomials possessing multiple roots with greater accuracy and reasonable efficiency than had previously been available. This general purpose algorithm is actually applicable to any class of polynomials with real coefficients. In essence, the algorithm can be divided into two separate stages, finding the real roots first, and then the complex roots. The first part comprises several procedures which first isolate a GCD sequence of unique polynomial factors containing distinct real roots, and then solves for these roots using a Sturm–Newton strategy. The second part isolates the complex factors by forming interpolating polynomials as an alternative to deflating the real roots, and then uses a Lehmer–Shur method to approximate initial root estimates for a complex Newton method. The composite algorithm was compared against [JT70] for real polynomials and yielded a significant increase in accuracy and reasonable speed for cases tested which included randomly generated polynomials with multiplicities from one to nine [Dun74]. Other related works which refer to this composite algorithm include [GK90, FL85, LN79, Mat79, JT75].

Madsen–Reid's Newton–Based Method

Madsen and Reid's Newton–based FORTRAN algorithms for both complex and real polynomial coefficients are included in [MR75]. Based on Madsen's earlier work [Mad73], they utilize a two stage global and local Newton step process to isolate and approximate all the zeros of a polynomial. To ensure stable deflation, the algorithm repeatedly approximates the zeros of the smallest modulus by first successively searching and computing tentative Newton–based steps until the iterate is close enough to apply straightforward Newton approximation to quadratically converge to the root. Error estimation methods expanded from Peters and Wilkinson [PW71] are implemented to enhance convergence criteria for both complex and real coefficient cases. The method reports a two–fold advantage over the Jenkins and Traub algorithm [JT70]: a 2–4 times increase in efficiency with at least as accurate results, and a simpler algorithm yielding less bulky code (object code about $\frac{2}{3}$ as long). The FORTRAN code is set up to easily accommodate various precisions, checks for leading and trailing zero coefficients, and normalizes deflated coefficients. Other works referring to this method include [Wes81, RR78, SR77, Moo76].

Laguerre's Method

Laguerre's method FORTRAN and C algorithms [PFTV86, PFTV90] find all the roots of a complex polynomial by first isolating guaranteed initial approximations for any root (real, complex, simple, and multiple) using parabolas, and then uses further refinement, or root polishing, techniques to converge to within required precision. Laguerre's method is guaranteed to converge independent of any initial values and incurs the calculation of $f(t)$, $f'(t)$, and $f''(t)$ for each stage of the iteration which provides cubic convergence near simple real roots and linear convergence near multiple zeros. Other works referring to this method include [Ric83, RR78, Act70, Hou70, Bod49].

2.5 Power Form Real Root Finders

This section reviews existing power form real root finding methods that may be categorized according to how they isolate the real roots of the polynomial. We consider methods based on:

1. Sturm sequences,
2. Differentiation,
3. Variation of signs,
4. Interval techniques.

2.5.1 Sturm Sequence Techniques

The determination of the number of real roots on an interval using Sturm sequences has been described above. One root–finding technique using Sturm sequences is described in [Dun72]. Another is the method of Hook and McAree.

Hook–McAree's Method

Hook and McAree present a C implementation using Sturm sequences to bracket the real roots of power form polynomials for subsequent refinement by the modified regula falsi method [HM90]. The isolation phase builds a Sturm sequence using a pseudo–polynomial remainder sequence derived from the original polynomial and its first derivative as outlined in [§ 2.2.2], and then applys a bisection technique to the sequence until a single sign variation is achieved. A ray tracing program used this algorithm to render algebraic surfaces of arbitrary order which frequently encounters the solution of ill-conditioned polynomials.

2.5.2 Differentiation Techniques

Polynomial derivative sequences provide natural bounding intervals for the real roots of the given polynomial, as exemplified by the work of Collins and Loos.

Collins–Loos' Method

Collins and Loos [CL76] present a polynomial real root isolation method that utilizes recursive derivative sequences of the given power form polynomial to achieve root interval separation, applying Rolle's theorem, a simple tangent construction heuristic, and polynomial greatest common divisor calculations to determine the existence of two or no roots in an interval. The algorithm is compared both analytically and empirically to Heindel's Sturm sequence algorithm [Hei71] which uses integer arithmetic to compute the real zeros of a polynomial. It performs significantly faster in general, which is attributed primarily to smaller coefficients in the derivative sequence versus the Sturm sequence. Test cases consisted of both random and perturbed random product polynomials as well as Chebyshev and Legendre polynomials up to degree 25. Other references that cite this work include [CA76, CL82, MC90, Rum79].

2.5.3 Variation of Signs Techniques

Real root isolation strategies that use Descartes' rule of signs to bound the roots are contained in the work by Collins and Akritas.

Collins–Akritas' Method

Collins and Akritas [CA76] present a polynomial real root isolation method that combines Descartes' rule of signs with linear fractional transformations. This is a modification of Uspensky's algorithm based on a theorem by Vincent [Usp48]. The algorithm is 2 to 8

times more efficient than the Collins and Loos method described above. Other sources that cite this work include [CL82, MC90].

2.5.4 Interval Techniques

Real root isolation schemes that construct bounding intervals to contain and exclude the roots include Hansen's Newton interval and Dedieu and Yakoubsohn's exclusion interval methods, respectively.

Hansen's Newton Interval Methods

Hansen develops polynomial real root finding methods [Han78b, Han78a] introduced by Moore [Moo66] that extend interval analysis to Newton's method. This guarantees isolation and approximation of all the real roots of a polynomial by bounding intervals. Other interval versions of Newton's method are cited in Hansen's works as well as in [AH83, Han69, Moo79]. Further sources extending these concepts to solutions of complex roots, as well as systems of non–linear equations, include [AH83, GH73, Han69, HG83, Moo77, Moo79].

Dedieu–Yakoubsohn's EXCLUSION Method

Dedieu and Yakoubsohn's exclusion algorithm [DY93] is a real root isolation scheme that localizes all the real roots of a real polynomial, based on an exclusion function which defines intervals on which the original polynomial does not have any roots. The arrangement of the exclusion intervals provides the necessary bounding intervals for approximation by an appropriate root bracketing scheme [§ 2.3.2,2.3.3]. The exclusion method guarantees convergence to the accuracy ε in $\mathcal{O}(|log\,\varepsilon|)$ steps and is stable under modifications of the initial iterate as well as rounding errors. The exclusion scheme proved stable for higher degree polynomials as opposed to the unstable effects of a Sturm's method with identical step convergence order, although Sturm's method exhibits better efficiency for well–conditioned

lower degree polynomial cases. The exclusion algorithm also extends to computing complex roots. The authors provide a C version of the isolating algorithm.

2.6 Bernstein Form Real Root Finders

This section briefly reviews existing Bernstein form polynomial real root finding methods, which are grouped according to how they isolate the real roots, using either:

1. Recursive subdivision,
2. Newton–based methods,
3. Hull approximation techniques.

2.6.1 Recursive Subdivide Techniques

Bernstein root finders that repeatedly subdivide to isolate and even approximate the real roots of a polynomial are designated as recursive subdivision techniques. Three methods developed by Lane and Riesenfeld, Rockwood, and Schneider are reviewed below. Another method is presented in this document in [§ 6.3] that utilizes subdivision and derivative heuristics to isolate each root. The roots are then further refined using a hybrid serial approximation scheme adapted to Bernstein form polynomials.

Lane–Riesenfeld's Method

Lane and Riesenfeld [LR81] couple recursive bisection with properties of the Bernstein form to isolate and eventually approximate real roots over a specified interval. The polynomial is recursively subdivided at its midpoint until either the root is approximated to a specified precision, or no root exists in the polynomial subinterval, whereupon the polynomial segment is eliminated. The variation diminishing property [§ 2.1.2] is utilized to indicate whether or not a root exists in the subinterval. Binary subdivision incurs $\mathcal{O}(n^2)$ steps and

provides one bit of accuracy for each subdivision step. Other related works which reference this method include [Gra89, MC90, RKF88, Roc89, Sch90a].

Rockwood's Method

Rockwood's algorithm [Roc89] is a variation on the Lane and Riesenfeld method suitable for parallel and vector processing. Instead of binary subdivision, Rockwood uses a step hueristic that estimates the root using the linearly interpolated value between two points on the curve receiving the greatest influence from the nearest control points to the crossing. This estimate eventually becomes a Newton step in the vicinity of the approximate root, exhibiting quadratic convergence. The ordered real roots are found to a specified tolerance. This work is also referenced in [MC90].

Schneider's Method

Schneider's algorithm [Sch90a] is also a variant of the Lane and Riesenfeld algorithm which uses recursive binary subdivision to approximate the ordered roots over the Bernstein interval. A root is determined when either the recursion depth limit is reached, or when the control polygon crosses the abscissa once and approximates a straight line, or is *flat enough*, the root in the corresponding region being then computed as the intersection of the t–axis with the chord joining the first and last control point. The *flat enough* termination criterion is achieved when the maximum perpendicular distance of the control points to the chord is bounded by a specified tolerance.

2.6.2 Newton–Based Techniques

Bernstein root finding schemes that use Newton-type steps to isolate and approximate the real roots of a polynomial are exemplified by the algorithms of Grandine and of Marchepoil and Chenin.

Grandine's Method

Grandine [Gra89] computes all the zeros of a univariate spline function using a Newton interval method that requires bounds on the derivative of the function, without breaking the interval into its individual polynomial pieces. The method exhibits Newton convergence for simple roots, and thus at best linear convergence for multiple roots.

Marchepoil–Chenin's Method

A report by Marchepoil and Chenin in French [MC90] uses a combination of Bernstein subdivision techniques combined with Newton iterations to find the ordered set of real roots over an interval for ray tracing applications. Unfortunately, the author was unable to obtain a complete translation of this report and thus cannot provide a detailed comparison with other methods. The method is benchmarked against the Rockwood and Lane–Riesenfeld algorithms described above.

2.6.3 Hull Approximation Techniques

Bernstein root finding methods can also exploit the convex hull property to isolate and even approximate the real roots of a polynomial. A method by Rajan, Klinkner, and Farouki exemplifies this type of method. Two other methods are presented in this document that use optimal convex hulls to provide steps that in the first case [§ 6.4] isolate the ordered real roots (which are subsequently approximated using a hybrid serial approximation method adapted to Bernstein form polynomials), and in the second case [§ 6.1] successively approximate the ordered real roots.

Rajan–Klinkner–Farouki's Method

Rajan et al. [RKF88] use *parabolic hulls* to isolate and approximate simple real roots of a Bernstein form polynomial. A parabolic hull is a parabolic generalization of the convex hull property [§ 2.1.2] that bounds the polynomial from above and below. This method favors high degree polynomials (examples up to degree 2048) with few roots over the interval of interest, and was up to several orders of magnitude faster than other algorithms for this type of problem. The scheme exhibits cubic convergence when approximating a root. Generalization of the technique to higher–degree hulls is also outlined.

Chapter 3

Review of Numerical Considerations

Robust computer implementation of polynomial root finding methods requires, in addition to a formal mathematical description of the algorithm, a keen awareness of the limitations of the floating–point number system and issues of numerical stability and error propagation. This chapter reviews key numerical considerations pertaining to the implementation of polynomial real root finding algorithms. Section [§ 3.1] reviews sources and types of numerical error noted in the literature. Section [§ 3.2] discusses general deterministic and indeterministic analysis techniques for estimating bounds for roundoff error. Section [§ 3.3] reviews numerical stability and condition properties including the condition of polynomial roots with respect to perturbations in the coefficients, with particular attention to the power and Bernstein representations. Section [§ 3.4] reviews and provides an example of the conversion process between Bernstein and power forms of a polynomial. Finally, section [§ 3.5] presents guidelines for benchmarking the performance of polynomial root finders, identifying various classes of polynomials appropriate for testing.

3.1 Numerical Error

The computer implementation of mathematical algorithms involves a variety of ways in which numerical errors may be generated, propagated, and amplified. This section outlines the various types and sources of numerical error discussed in the literature [Buc66, Dod78, Dun72, Hen82, Hil74, Hou53, McN73, Mor83, Pen70, PC86, Piz75, RR78, Sta70] with particular emphasis on floating–point roundoff error, indicating guidelines to minimize the error whenever possible.

3.1.1 Sources and Types of Numerical Error

We may distinguish between two distinct types of error, which arise in the the "modeling" and "computing" stages:

Modeling Error (as distinct from numerical error) reflects the deficiencies of, for example, a discrete approximation to a continuous mathematical problem, or the possible failure of an iterative method to have guaranteed convergence to the desired solution.

Computational Error measures the numerical errors due to implementing an algorithm in floating–point arithmetic on a computer. Computational error may be classified into the following categories:

Gross Error is caused by mistakes, oversights, or malfunctions of man or machine.

Truncation Error results from using a finite approximation to represent an ideal function, equation, or value that does not have a tractable finite representation.

Convergence Error arises from either a divergent iterative process, or premature termination of a convergent process.

Overflow-Underflow Error results from the finite dynamic range of the floating–point number system. It may be largely avoided by using appropriate data scaling and normalization techniques.

Roundoff Error arises from the fact that the results of floating–point operations must be rounded to be representable within the floating–point number system. This incurs errors at each arithmetic step of an algorithm, and the propagation and possible amplification of errors incurred in preceding steps.

Although distinct from computational error, the proper modeling and formulation of a mathematical problem is an important aspect of robust implementations [§ 3.3]. As a general guideline for minimizing numerical error, computations should be kept as concise as possible, although there are exceptions to even this rule.

Causes of Computational Error

Computational errors arise under a variety of circumstances, such as:

1. *Inherent error* in the input data of a computation. This may arise from physical measurement errors, the fact that the input data is itself the result of floating–point computations, or simply the need to round "exact" input to floating–point format.
2. *Cancellation error* in subtracting nearly equal numbers, which incurs an amplification of pre–existing relative errors in the operands. This may be ameliorated by rearranging sums so that differences close to zero are avoided, and using *reducible subtraction operations* [Vig78] which detect and ensure the summation of like signed numbers.
3. Roundoff in adding or subtracting one large and one small number. Summing numbers in increasing magnitude helps remedy this source of error.
4. Simple roundoff due to the ordinary arithmetic operations of addition, subtraction, multiplication, and division.
5. Floating–point overflow or underflow (e.g., due to division by a very small number).

Other pertinent tips for minimizing roundoff error include:

- Avoid (if possible) expressions of the form a^b where b is a floating point number,
- Use double precision for floating–point calculations,
- Rearrange formulas to use original data rather than derived data,
- Rearrange formulas to minimize arithmetic operations,
- Avoid chaining operations which involve round–off error, and
- Work with integers whenever possible and economically feasible.

3.1.2 Roundoff Error

Discussions regarding floating-point computation preface most numerical analysis texts in part and is exclusively defined and examined by Sterbenz [Ste74]. In addition, a text by Plauger [Pla92] provides extremely useful insight regarding standardized floating-point constants for a UNIX-based machine usually located in the file: *\usr\include\float.h*.

Floating-Point Computation

A computer expression in $\mathcal{F}$ formed by the arithmetic set of computer operators

$$\{ \oplus, \ominus, \odot, \oslash \} \tag{3.1.2.1}$$

that numerically approximates an algebraic expression defined by the set of real numbers $\mathcal{R}$ and formed by an arithmetic set of exact operators

$$\{ +, -, \cdot, / \} \tag{3.1.2.2}$$

results in numerical *rounding* of the algebraic expression when $\mathcal{F}$ and its respective operators define the set of *floating-point* numbers and operations. A floating-point number x has the form

$$x = \pm m \ \beta^e \in [x_{min}, x_{max}] \in \mathcal{F} \tag{3.1.2.3}$$

where the signed number $m \in \mathcal{F}$ is the *normalized mantissa* of x, having d *significant digits* in the *base* β of the machine, and $e \in \mathcal{Z}$ is the *exponent* of x, lying within some range $[e_{min}, e_{max}]$.

Machine Epsilon ϵ and Roundoff Unit η

The precision of floating-point arithmetic is characterized by the *machine epsilon* which is defined as the smallest floating point number ϵ such that

$$1 \oplus \epsilon > 1 \qquad or \qquad \epsilon = \beta^{1-d}, \tag{3.1.2.4}$$

and its corresponding *roundoff error bound* by the *machine roundoff unit* [Mor83] which is defined as the smallest floating point number η such that

$$1 \oplus \eta \;=\; 1 \qquad or \qquad \eta \;=\; \frac{1}{2}\,\epsilon \;=\; \frac{1}{2}\,\beta^{1-d} \tag{3.1.2.5}$$

which reduces to $\eta = 2^{-d}$ on a binary machine.

A function that calculates the machine epsilon for any machine is given in the following algorithm [FMM77].

Algorithm 3.1.2.1 (ϵ = ComputeMachineEps ( )) *Compute and return the machine epsilon* ϵ.

1. $\epsilon = 1$
2. While ($(1+\epsilon) > 1$)
3. $\quad \epsilon = \frac{1}{2}\epsilon$

   EndWhile

End of ComputeMachineEps

Roundoff Error Due to Floating-Point Operations

The *absolute error* of a computed value x due to roundoff of a binary floating-point operation $\{ \oplus, \ominus, \odot, \oslash \}$ is represented by

$$\varepsilon_{x_\eta} \tag{3.1.2.6}$$

and the corresponding *relative error* is denoted by

$$\frac{\varepsilon_{x_\eta}}{x} \tag{3.1.2.7}$$

which measures the *precision*, or the number of correct digits, in x. The formulas that account for the roundoff error due to floating-point operations are

$$\begin{aligned} x \;\oplus\; y &= (x+y)\,(1+\delta) \\ x \;\ominus\; y &= (x-y)\,(1+\delta) \\ x \;\odot\; y &= x \cdot y\,(1+\delta) \\ x \;\oslash\; y &= x/y\,(1+\delta) \end{aligned}$$

where δ is the relative roundoff error introduced by a single floating-point operation which satisfies the relation $|\delta| \leq \eta$ [Dod78, Hil74, Hou53, IK66, KM81, Pen70, PW71, PW83, Ric83, Sta70]. Note that these formulas do not take into consideration inherent or pre–existing errors in the operands (see [§§ 3.2.2],[PW83]).

3.2 Estimating Bounds for Roundoff Error

There are a variety of approaches to monitoring the propagation of rounding errors in numerical computations so as to estimate error bounds for computed quantities. They may be boradly classified as as either *deterministic* or *indeterministic* techniques [BC86, FV85, Ham71, Hen82, Mil75b, RR78, Ric83, Ste74, Wil63].

Deterministic techniques estimate an *upper bound* for the roundoff error incurred during a computation, and may be divided into the following approaches:

1. Forward Error Analysis
2. Running Error Analysis
3. Backward Error Analysis
4. Interval Error Analysis

Indeterministic techniques estimate an *average bound* based on certain measures of reliability for the roundoff error incurred during a computation which separates into the following current approaches.

1. General Stochastic Error Analysis
2. Permutation-Perturbation Error Analysis

It is important to note that these methods yield estimates of the propagated roundoff error at the expense of a loss of efficiency in both execution and a more complicated implementation. The general purpose polynomial root finder implemented by Madsen and Reid discussed in [§§§ 2.4] reported their running error analysis overhead increased CPU time usually about 25% for simple roots and up to 100% when many multiple roots were considered [MR75]. Thus, a good rule of thumb is to use methods that avoid roundoff error analysis when possible, and to use error analysis judiciously when needed.

3.2.1 Forward Error Analysis

Forward error analysis bounds the absolute or relative roundoff errors of the intermediate steps of a computation. Various perspectives on forward error analysis are presented in [Mil75b, PW71, RR78, Ric83, Ste74, Wil63]. This approach is often too conservative, yielding error bounds that can be many orders of magnitude larger than the actual error and thus virtually useless when used as a convergence criterion. In addition, forward error analysis is tedious and often difficult to apply to complex computations (although applied skillfully to straightforward equations this approach can produce a tight error bound on the computed value). Also, it does not account for cancellation effects — see [§ 4.2.1] below.

3.2.2 Running Error Analysis

Peters and Wilkinson coined the term *running error analysis* in [PW71] to describe a method that computes error estimates "on–the–fly" for the case of Horner evaluation of $f(x)$. The essence of this method is to obtain for each intermediate step of the computed value an absolute or relative bound on the error due to the effects of both the inherent and roundoff error. Such methods are often termed *linearized running error analyses*, since they neglect higher–order terms in small quantities. Works that discuss concepts related to running error analysis include [Hen64, Hen82, Hil74, Hou53, Mor83, Pen70].

The *absolute error* of a computed value x represented by

$$\varepsilon_x \;=\; \varepsilon_{x_\eta} \;+\; \varepsilon_{x_\iota} \tag{3.2.2.8}$$

that accounts for both roundoff and inherent error when computed with a binary operator consists of a component ε_{x_η} due to roundoff from the a floating point operation [§§§ 3.1.2] as well as a component ε_{x_ι} due to the inherent error which already exists in x due to prior sources of error [PW83]. The corresponding *relative error* is measured and denoted by

$$\frac{\varepsilon_x}{x} \;=\; \frac{\varepsilon_{x_\eta}}{x} \;+\; \frac{\varepsilon_{x_\iota}}{x}. \tag{3.2.2.9}$$

There are two main approaches presented in the literature that account for the inherent and roundoff error terms in equations (3.2.2.8) and (3.2.2.9).

Binary Operator approach is based on the fact that variables in any sequence of a computation are only processed two at a time by binary operators. Thus, the operations composing the entire computation are arranged into a sequence of binary operations.

The error bound is obtained by accumulating the errors from each binary operation such that the effects of each binary operation contributes to the next and so forth.

A calculation graph which maps the two operands of each binary operation to the corresponding nodes of a binary graph is presented in [DM72].

Partial Derivative Operator approach is based on the idea that a computation can be expressed as the sum of a sequence of linear successive operators. A partial differential equation represents the sum of these linear successive operators which each represent an independent variable in the differential equation.

The error bound is obtained by applying the chain rule to the partial differential equation for each linear successive operator, and thus, is the sum of the effects of all the errors contributing to the computation.

A calculation graph which maps each partial derivative operation to a corresponding node of a directed graph is presented in [Pen70, PW83].

Both methods work for simple expressions. Complex formulations need to be decomposed or factored into simple parts which are then composed and analyzed as a simple function. Although running error analysis provides reasonably tight bounds, it at least doubles the operations in solving a particular expression.

3.2.3 Backward Error Analysis

Rather than tracking the absolute or relative errors of computed values at each intermediate step of an algorithm, backward error analysis is concerned with showing that the computed result is *exact* for some "neighboring" problem, corresponding to perturbed input parameters. The approach indicates a range of input values which give results differing from the true solution by an amount that reflects the cumulative effects of intermediate computational errors. Sources describing this process include [Hen82, Mil75b, Mil75a, RR78, Ric83,

Ste74, Wil63]; see also [§ 4.2.3] below.

3.2.4 Interval Error Analysis

Interval error analysis introduced by Moore [Moo66] and also covered by [AH83, GL70, Han69, KM81, Moo66, Moo79, RL71, Ste74] replaces each computed value x with a scalar interval represented by

$$[a,b] \;=\; \{x \mid a \le x \le b\}. \tag{3.2.4.10}$$

Interval arithmetic operations for two intervals $[a,b]$ and $[c,d]$ are represented by

$$\begin{array}{lcll} [a,b] & + & [c,d] & = & [\; a+c,\; b+d \;] \\ [a,b] & - & [c,d] & = & [\; a-d,\; b-c \;] \\ [a,b] & \cdot & [c,d] & = & [\; \textsc{Min}(ac,ad,bc,bd),\; \textsc{Max}(ac,ad,bc,bd) \;] \\ [a,b] & / & [c,d] & = & [\; a,\; b \;] \cdot [\; 1/d,\; 1/c \;] \end{array}$$

where interval division is defined only for denominator intervals that do not contain 0.

Although interval analysis gives a guaranteed bound the error of the computed value, this bound is often too conservative and can be very expensive due to interval arithmetic overhead if not judiciously applied [Ham71]. Interval analysis has been applied to algorithms involving Bernstein–form computations [Rok77, SF92], bounds on interval polynomials [Rok75, Rok77], and the determination of polynomial zeros [GH72, Han70].

3.2.5 General Stochastic Error Analysis

Stochastic error analysis techniques estimate the correctness of a computation by studying the statistical variance and mean values for the generated roundoff error. These values are not upper (worst–case) error bounds, but rather approximations to an upper error bound to within a certain level of confidence. Various concepts and techniques for the general application of this process are presented in [Buc66, Ham71, Hil74, Hen64, Hou53, KL73, Pen70, Piz75, RR78, Ric83, Ste74, Wil63]. Of well–tested approach is the permutation–perturbation stochastic method, which is reviewed in the next subsection.

3.2.6 Permutation–Perturbation Error Analysis

The permutation–perturbation or CESTAC technique of error analysis Introduced by Vignes and La Porte [VL74] and further developed in [FV85, Vig78] determines the precision of a floating–point computation by considering the stochastic effects that roundoff due to the permutation effects of the computer arithmetic operations and the perturbation of their possible images has on the computed values. The method essentially simplifies to the following three steps which, provide an error estimate with a computed number of significant digits with a probability of 95% as a consequence of Student's law and the central limit theorem [Alt86].

1. Compute three values $\{x_i\}_{i=1}^{3}$ via an (machine dependent) algorithm that randomly permutes the permutable operators and randomly perturbs the last bit of the mantissa on any intermediate value.

2. Compute the mean value $\bar{x}$ and the standard deviation σ of the three values $\{x_i\}$ by

$$\bar{x} = \frac{\sum_{i=1}^{3} x_i}{3} \qquad and \qquad \sigma = \sqrt{\frac{\sum_{i=1}^{3}(x_i - \bar{x})^2}{2}}.$$

3. Compute the number of significant digits $\bar{d}$ of $\bar{x}$ from the formula

$$\bar{d} \;=\; \log_{10} \frac{\bar{x}}{\sigma}. \tag{3.2.6.11}$$

Although the computational overhead is increased by at least a factor of three (step 1), this method provides an extremely accurate estimate of the incurred roundoff error. This method has been successfully applied to algorithms involving computational geometry [DS90], eigenvalues [Fay83], Fourier transforms [BV80], integration of ordinary differential equations [Alt83], linear and non–linear programming [Tol83], optimization [Vig84], parallel and closed form polynomial root finding [Alt86, Vig78]. It has been recommended for scientific computing in general in [BC86, Vig88], with sample FORTRAN codes provided in [BV80, Vig78] of the machine dependent permutation–perturbation function required in the initial step above.

3.3 Numerical Stability and Condition

An important indication of whether or not computational errors are likely to be significant in a given problem is the *condition* or "intrinsic stability" of that problem. If small changes in the input values induce large changes of the (exact) solution, the problem is said to be ill–conditioned, and it is unrealistic to expect high–accuracy solutions from finite–precision computations. A related, but distinct, issue is whether the particular algorithm used to address a given problem is stable (e.g., whether it incurs subtraction of near–identical quantities). This section reviews some basic definitions and concepts pertaining to numerical stability and conditioning, the condition of perturbing power form polynomial coefficients, and compares the condition of the Bernstein and the power polynomial forms.

3.3.1 Preliminary Definitions

We summarize below the basic concepts and terminology concerning numerical stability and condition [FR87, Hen82, RR78, Ric83].

Numerical Stability is a property of an algorithm which measures its propensity for generating and propagating roundoff error and inherent errors in the input data.

Numerical Instability results when the intermediate errors due to roundoff strongly influence the final result.

Numerical Condition is a mathematical property of a problem which measures the sensitivity of the solution to perturbations in the input parameters.

Condition Number is one or more scalar values that describes the condition of a problem, estimating how much uncertainty in the initial data of a problem is magnified in its solution; condition numbers may be formulated in terms of both absolute and relative perturbations of the input and output values.

Well–Conditioned describes a problem characterized by a small condition number, i.e., changes of the initial data yield commensurate changes of the output data, so the problem can be solved to reasonable accuracy in finite–precision arithmetic.

Ill–Conditioned describes a problem characterized by a large condition number, changes of the input leading to much larger changes of the output, so that accurate solutions

are difficult to compute in finite–precision arithmetic.

Thus, it is almost impossible for an ill–conditioned problem to be numerically stable in implementation [Ric83]. On the other hand, an unstable method for a well–conditioned problem may at first produce accurate results until the propagation of roundoff accumulates to yield erroneous intermediate and final results [RR78]. Since the condition of a problem is dependent upon its formulation, it may be possible to ameliorate effects of ill–conditioning by reformulating the problem, i.e., restating it in terms of different input/output variables [Ric83].

3.3.2 Condition of Perturbing Polynomial Coefficients

The numerical condition of polynomials with respect to root–finding is analyzed by perturbing the coefficients and observing the resulting effects on the roots. This is accomplished by establishing a relationship between the roots $\{\alpha_i\}$ of the original polynomial

$$f(x) \;=\; \sum_{i=0}^{n} a_i x^i$$

and the roots $\{\alpha_i + \delta\alpha_i\}$ of the perturbed polynomial

$$f(x) \;=\; \sum_{i=0}^{n} (a_i + \delta a_i) x^i \qquad (3.3.2.12)$$

where for purposes of analysis the coefficient errors $\{\delta a_i\}$ are regarded as begin specified arbitrarily (in practice they might reflect rounding errors due to earlier computations).

For instance, the well–known Wilkinson polynomial [Wil63] represented in product form by

$$f(x) \;=\; \sum_{i=1}^{20}(x+i), \qquad \{\alpha_i\} = \{-1, -2, \ldots, -20\} \qquad (3.3.2.13)$$

yields the following roots $\{\alpha_i + \delta\alpha_i\}$ (correct to 9 decimal places):

-1.00000 0000		
-2.00000 0000		
-3.00000 0000		
-4.00000 0000		
-4.99999 9928		
-6.00000 6944		
-6.99969 7234		
-8.00726 7603		
-8.91726 0249		
-10.09526 6145	$\pm$	0.64350 0904 i
-11.79363 3881	$\pm$	1.65232 9728 i
-13.99235 8137	$\pm$	2.51883 0070 i
-16.73073 7466	$\pm$	2.81262 4894 i
-19.50243 9400	$\pm$	1.94033 0347 i
-20.84690 8101		

when the power coefficient a_{19} is perturbed by as little as $\delta a_{19} = 2^{-23}$ [Hou70, PC86, Piz75, RR78, Wil63]. These results illustrate the difficulties in computing the roots of high degree polynomials represented in power form.

3.3.3 Condition of Bernstein and Power Forms

Both analytical [FR87, FR88] and empirical [SP86] studies indicate that polynomials expressed in Bernstein form can provide superior computational stability and root conditioning than in the power form. The following results outlined in [FR87] regarding the improved root condition of the Bernstein basis over the power basis are duplicated for convenience:

Given a Bernstein basis defined over an arbitrary interval $[a, b]$ (typically [0,1]) that contains all the roots of interest, and a power basis defined about any point (typically the origin) excluding the interior of $[a, b]$, then for any simple real root of an arbitrary polynomial the root condition number

1. is smaller in the Bernstein basis than in the power basis,
2. decreases monotonically under Bernstein degree elevation,
3. decreases monotonically under Bernstein subdivision, and

4. is smaller in the Bernstein basis than in any other basis which may be expressed as a non-negative combination of the Bernstein basis on that interval.

The same holds true for multiple roots if the condition number (which is formally infinite) is regarded as the constant factor of the non-linear growth rate of the root displacements with the coefficient perturbations.

3.4 Conversion Between Bernstein and Power Forms

Consider a polynomial in Bernstein form over the $[0,1]$ domain

$$B(t) = \sum_{i=0}^{n} b_i \binom{n}{i} (1-t)^{n-i} t^i$$

and the same polynomial in power form

$$P(t) = \sum_{i=0}^{n} p_i t^i.$$

The problem we consider is, given the Bernstein coefficients b_i, find the corresponding power coefficients p_i (Bernstein to power basis conversion) or given the p_i, find the b_i (power to Bernstein conversion).

An elegant solution to this problem can be obtained by performing a Taylor's series expansion of $B(t)$:

$$B(t) \equiv B(0) + B'(0)t + \frac{B''(0)t^2}{2!} + \ldots + \frac{B^{(n)}(0)t^n}{n!}.$$

A power basis polynomial is equivalent to a Bernstein basis polynomial ($P(t) \equiv B(t)$) if and only if

$$\frac{P^{(i)}(0)t^i}{i!} \equiv \frac{B^{(i)}(0)t^i}{i!}, \quad i = 0, \ldots, n.$$

But, for the power basis,

$$\frac{P^{(i)}(0)}{i!} = p_i$$

so

$$p_i = \frac{B^{(i)}(0)}{i!},\ i = 0, \ldots, n. \tag{3.4.0.14}$$

The terms $B^{(i)}(0)$ can be found as follows. Recall from Section 5.6 that the coefficients of the derivative of a polynomial in Bernstein form are:

$$n(b_1 - b_0),\ n(b_2 - b_1),\ \ldots,\ n(b_n - b_{n-1}).$$

The coefficients of the second derivative are:

$$n(n-1)(b_2 - 2b_1 + b_0),\ n(n-1)(b_3 - 2b_2 + b_1),\ \ldots, i\ n(n-1)(b_n - 2b_{n-1} + b_{n-2}).$$

Since $B(0) = \sum_{i=0}^{n} b_i \binom{n}{i}(1-0)^{n-i}0^i = b_0$, we have

$$B'(0) = n(b_1 - b_0), \quad B''(0) = n(n-1)(b_2 - 2b_1 + b_0)$$

$$B^{(i)}(0) = n(n-1)\cdots(n-i+1)\sum_{j=0}^{i}(-1)^{(i-j+1)}\binom{i}{j}b_j.$$

This can be written more neatly if we define the recurrence

$$b_i^j = b_{i+1}^{j-1} - b_i^{j-1}$$

with $b_i^0 \equiv b_i$. Then

$$B^{(i)}(0) = n(n-1)\cdots(n-i+1)b_0^i = \frac{n!}{(n-i)!}b_0^i.$$

From equation 3.4.0.14,

$$p_i = \frac{n!}{(n-i)!i!}b_0^i = \binom{n}{i}b_0^i.$$

Thus, the problem reduces to one of finding the values b_0^i. This is easily done using a difference table:

$b_0^0 = b_0 = p_0$	$b_1^0 = b_1$	$\ldots$	$b_n^0 = b_n$
$b_0^1 = b_1^0 - b_0^0 = p_1/\binom{n}{1}$	$b_1^1 = b_2^0 - b_1^0$	$\ldots$	$b_n^0 = b_n^0 - b_{n-1}^0$
$b_0^2 = b_1^1 - b_0^1 = p_2/\binom{n}{2}$	$b_1^2 = b_2^1 - b_1^1$	$\ldots$	$b_n^2 = b_n^1 - b_{n-1}^1$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$b_0^{n-1} = b_1^{n-2} - b_0^{n-2} = p_{n-1}/\binom{n}{n-1}$	$b_1^{n-1} = b_2^{n-2} - b_1^{n-2}$		
$b_0^n = b_1^{n-1} - b_0^{n-1} = p_n$			

Thus, to perform Bernstein to power basis conversion, load the Bernstein coefficients into the top row and compute the difference table. Scale the left column by $\binom{n}{i}$, and you have the power coefficients.

To perform power to Bernstein conversion, divide the power coefficients by $\binom{n}{i}$, load them into the left column, compute the difference table backwards, and read the Bernstein coefficients off the top row.

3.4.1 Example

Convert to power basis the degree 4 Bernstein form polynomial with coefficients $(1, 3, 4, 6, 8)$. This is done by setting up the difference table

1	3	4	6	8
2	1	2	2	
-1	1	0		
2	-1			
-3				

so the power coefficient are taken from the left column, times the binomial coefficients:

$$\begin{aligned} p_0 &= 1\binom{4}{0} = 1 \\ p_1 &= 2\binom{4}{1} = 8 \\ p_2 &= -1\binom{4}{2} = -6 \\ p_3 &= 2\binom{4}{3} = 8 \\ p_4 &= -3\binom{4}{4} = -3 \end{aligned}$$

3.4.2 Closed Form Expression

The conversion from Bernstein to power basis can be written concisely as follows:

$$p_i = \sum_{k=0}^{i} b_k \binom{n}{i} \binom{i}{k} (-1)^{i-k}.$$

Power to Bernstein conversion is accomplished by the formula:

$$b_i = \sum_{k=0}^{i} \frac{\binom{i}{k}}{\binom{n}{k}} p_k. \tag{3.4.2.15}$$

3.4.3 Numerical Ramifications of Basis Conversion

The numerical condition of basis transformation has been studied in [FR88, DD88, Far91b]. The condition number for the basis transformation matrix gives an upper bound — for arbitrary polynomials — on the largest possible value by which basis conversion amplifies the relative coefficient error. It grows exponentially with the polynomial degree n.

3.5 Performance of Polynomial Root Finders

Guidelines for performance comparisons of mathematical software are proposed in [CDM79, Ign71, Ric83], and more specifically for polynomial root finding algorithms by Jenkins and Traub in [JT74, JT75] and Wilkinson in [Wil63]. This section reviews principles regarding establishing and evaluating the overall performance of polynomial root finding algorithms based on the above references.

3.5.1 Benchmarking Principles

The criteria for testing root finding algorithms may be concisely summarized under the categories: 1) Robustness, 2) Convergence, 3) Deficiencies, and 4) Assessment.

Program Robustness is validated with pathological examples that test for specific program properties and underlying methods, such as:

1. Checking leading coefficients that approximate zero,
2. Checking trailing coefficients that approximate zero,
3. Proper handling of low degree polynomials,
4. Proper scaling of polynomial coefficients to avoid floating-point overflow and underflow,
5. Assessing the condition of each zero and adjusting the precision of computation accordingly,
6. Specifying iteration limits, and
7. Providing a failure exit or testable failure flags.

Program Convergence is a computational problem and needs validation regarding the:

1. Starting conditions for proper convergence,
2. Convergence of iterate sequences to acceptable tolerances,
3. Slow convergence of iterate sequences as in multiple zeros cases,
4. Divergence of iterate sequences, and
5. Program termination criteria.

Program Deficiencies for specific known implementation defects or weaknesses are important to document and are tested by forcing:

1. A program to produce an undefined result,
2. The decision mechanisms of the program to fail, and
3. Roundoff error to destroy accuracy of solution as in deflating zeros.

Program Assessment regarding performance is best analyzed by comparing program statistical information based on the establishment of meaningful measures.

1. **Reliability** of the solution.
2. **Accuracy** or closeness to the exact solution.
3. **Precision** or the number of significant digits in the answer.
4. **Efficiency** measurements including:
   - **CPU time** based on accurate timers and uniformly optimized code,
   - **Function evaluations**,
   - **Arithmetic operations**, and
   - **Storage requirements** regarding arrays, stacks, object code size.
5. **Portability** of the program to different computer platforms.

Polynomial Classes for Performance Assessment

Classes of polynomials for performance assessment should include both well-conditioned and ill-conditioned polynomials. A few well-conditioned cases should be tested against other algorithms in the literature, with the major testing focusing on ill-conditioned cases.

1. **Well-conditioned Polynomials**

Uniformly Modulated Zeros yield **similar zero cases**.

Polynomials constructed from uniformly distributed random zeros differ little from each other because the randomness tends to 'average out' in the coefficients.

Uniformly Modulated Coefficients yield **uniformly distributed zeros**.

Polynomials constructed from uniformly distributed random coefficients tend to have their zeros uniformly distributed around the origin near the unit circle.

2. **Ill-conditioned Polynomials**

Dual Bands of Modulated Zeros yield **clustered zeros**.

Polynomials constructed from both a wide band and a much narrower band of uniformly distributed random zeros produce ill-conditioned polynomials with clustered zeros, where the severity of the condition is controlled by the amount the narrower band is decreased and its percentage of zeros is increased.

Widely Modulated Zeros yield **widely varying sizes of zeros**.

Polynomials constructed from zeros whose mantissa and exponent are chosen from separate random uniform distributions yield zeros of widely varying size which often illustrate inefficiences in a program usually not apparent from other tests, e.g., poorly chosen initial iterates may tend to 'wander' slowly to the area of the zero and such behavior will be magnified on this class of polynomials.

Widely Modulated Coefficients yield **widely varying intervals of zeros**.

Polynomials constructed from coefficients whose mantissa and exponent are chosen from separate random uniform distributions yield zeros with widely varying intervals which typify wide applicability of the program.

Sources that define examples of the various generated polynomials include [Dun72, GK90].

Chapter 4

Preconditioning for Bernstein form polynomials

As mentioned in Chapter 1, several fundamental geometrical problems that arise in the processing of free–form curves and surfaces may be reduced computationally to the task of isolating and approximating the distinct real roots of univariate polynomials on finite intervals. Representative examples are: ray–tracing of surfaces [Han83, Kaj82, SA84]; computing intersections of plane curves [GSA84, SAG85, SP86]; finding a point on each loop of the intersection curve of two surfaces [Far86]; and the "trimming" of offset and bisector curves [FJ94, FN90a]. Such an approach is attractive in that it provides an algebraically precise formulation of the geometrical problem under consideration — given an "algorithmic" root finder — but it is widely perceived to be impractical in all but the simplest cases, due to the potentially poor numerical condition of the high–degree polynomials incurred. Thus, it has not found as widespread application as *subdivision methods* [CLR80, LR80] based on successively refined piecewise–linear approximations of curves and surfaces (generated recursively from their Bézier/B–spline control polygons).

In the aforementioned problems, the polynomials whose roots we seek are not specified *ab initio*, but rather must be "constructed" by a sequence of floating–point arithmetic operations on given numerical data, which may incur significant errors in the final polynomial

coefficients. Our investigations suggest that the *sensitivity of the roots to errors in the polynomial coefficients incurred during the construction stage* typically dominates any error associated with the actual root–finding process.

In this chapter we describe an approach to processing polynomials in Bernstein form that can significantly improve the accuracy of computed roots by minimizing adverse effects of the polynomial "construction" errors. In essence, the approach amounts to performing de Casteljau subdivision in the *earliest stages* of an algorithm, a procedure that we shall call "*a priori* subdivision." As is well known [FR87], the Bernstein form exhibits monotonically–decreasing root condition numbers with respect to subdivision. The practical meaning of this statement is that, if Bernstein representations are available *to the same relative accuracy in the coefficients* on both the nominal interval $[\,0,1\,]$ and a subset $[\,a,b\,]$ thereof, then the latter form always exhibits smaller worst–case errors for those roots actually located on the smaller interval.

Obviously, explicit floating–point subdivision of a polynomial with given initial coefficient errors cannot result in any diminution of the perturbations of its roots due to those initial errors — one cannot gain "something for nothing." Our key observation here, however, is that the consequences of errors incurred in the polynomial constructions for various geometrical operations can be ameliorated by supplying them with input polynomials that have been subdivided — *even in floating–point arithmetic* — a priori.

4.1 An Illustrative Example

A simple experiment on a well–known pathological example [Wil59] serves to motivate the proposed approach. Consider the degree–n Wilkinson polynomial having evenly–spaced roots k/n, $k = 1, \ldots n$, between 0 and 1. In the power basis, we have

$$P^n(t) \;=\; \prod_{k=1}^{n} (t - k/n) \;=\; \sum_{k=0}^{n} p_k \, t^k \,. \qquad (4.1.0.1)$$

Here the polynomial $P^n(t)$ is "constructed" by explicitly multiplying out, in floating–point arithmetic, the linear factors in the above product expression to obtain the power coefficients $\{p_k\}$. Column 2 of Table 4.1 shows the results of attempting to approximate the roots in the case $n = 25$ using this "constructed" power representation. The cause of the very large inaccuracies in the computed roots is discussed in [Wil59]. A simple intuitive understanding may be gleaned from the following calculation, taken from [Far91a], for the value of the polynomial halfway between the roots $t = 0.50$ and $t = 0.55$ in the case $n = 20$ (the values are correct to the number of digits shown):

$$
\begin{aligned}
p_0 &= +0.000000023201961595 \\
p_1\, t &= -0.000000876483482227 \\
p_2\, t^2 &= +0.000014513630989446 \\
p_3\, t^3 &= -0.000142094724489860 \\
p_4\, t^4 &= +0.000931740809130569 \\
p_5\, t^5 &= -0.004381740078100366 \\
p_6\, t^6 &= +0.015421137443693244 \\
p_7\, t^7 &= -0.041778345191908158 \\
p_8\, t^8 &= +0.088811127150105239 \\
p_9\, t^9 &= -0.150051459849195639 \\
p_{10}\, t^{10} &= +0.203117060946715796 \\
p_{11}\, t^{11} &= -0.221153902712311843 \\
p_{12}\, t^{12} &= +0.193706822311568532 \\
p_{13}\, t^{13} &= -0.135971108107894016 \\
p_{14}\, t^{14} &= +0.075852737479877575 \\
p_{15}\, t^{15} &= -0.033154980855819210 \\
p_{16}\, t^{16} &= +0.011101552789116296
\end{aligned}
$$

$$
\begin{aligned}
p_{17}\,t^{17} &= -0.002747271750190952 \\
p_{18}\,t^{18} &= +0.000473141245866219 \\
p_{19}\,t^{19} &= -0.000050607637503518 \\
p_{20}\,t^{20} &= +0.000002530381875176 \\
\hline
P^{20}(t) &= \quad 0.000000000000003899\ . \qquad (4.1.0.2)
\end{aligned}
$$

We see that the value of $P^{20}(t)$ is an *extremely small residual* resulting from the addition of large terms of alternating sign. Owing to massive cancellations, the final value is $\sim 10^{13}$ times smaller than the individual terms $p_k t^k$, and thus any initial relative errors in the coefficients p_k (as incurred, say, by "constructing" them in floating–point) will be *magnified* by this enormous factor in determining the relative error of the value $P^{20}(t)$ — resulting in a commensurate loss of accuracy in the root–finding process.

The above argument may be cast in more rigorous terms by computing the *root condition numbers* [FR87] for the polynomial $P^{20}(t)$, which measure the displacement of roots due to given fractional errors in the coefficients — these condition numbers are, in fact, of order 10^{13}. One should subsequently bear in mind the lesson of this example, namely, that severe ill–conditioning generally arises from the "magnification" of initial (perhaps seemingly innocuous) relative coefficient errors through cancellation effects.

4.1.1 The Bernstein Form

We mention the power form of the Wilkinson polynomial only as a point of reference; henceforth we shall be concerned solely with the Bernstein representation

$$B^n_{[0,1]}(t) \;=\; \prod_{k=1}^{n} \left[\, k(1-t) + (k-n)t \,\right] \;=\; \sum_{k=0}^{n} b_k \binom{n}{k} (1-t)^{n-k} t^k \qquad (4.1.1.3)$$

of this polynomial. Note that, apart from a multiplicative constant, each of the factors $k(1-t)+(k-n)t$ corresponds to the terms $t - k/n$ in (4.1.0.1). As emphasized in [FR87],

exact root	computed root of $P^{25}(t)$	computed root of $B^{25}_{[0,1]}(t)$	computed root of $B^{25}_{[.25,.75]}(u)$
0.04	0.039999999999997	0.0400000000000001	——
0.08	0.079999999997924	0.0799999999999781	——
0.12	0.120000000039886	0.1200000000006346	——
0.16	0.159999999495945	0.1599999999960494	——
0.20	0.200000020778928	0.1999999999971261	——
0.24	0.239999230565953	0.2400000000904563	——
0.28	0.280016180829233	0.2799999998181714	0.28000000000000000
0.32	0.319793958664642	0.3199999991463299	0.31999999999999999
0.36	0.362001308252336	0.3600000057566014	0.35999999999999979
0.40	0.391605636801344	0.3999999834411160	0.40000000000000060
0.44	complex	0.4400000307234767	0.44000000000000140
0.48	complex	0.4799999585120591	0.47999999999998849
0.52	complex	0.5200000426509138	0.52000000000002370
0.56	complex	0.5599999668961838	0.55999999999997959
0.60	complex	0.6000000183938222	0.60000000000000710
0.64	complex	0.6399999934951202	0.63999999999999929
0.68	complex	0.6800000009245819	0.68000000000000000
0.72	complex	0.7200000003141216	0.72000000000000000
0.76	complex	0.7599999998045576	——
0.80	complex	0.8000000000422774	——
0.84	complex	0.8399999999963719	——
0.88	complex	0.8799999999999752	——
0.92	0.925606820606031	0.9200000000000135	——
0.96	0.965436705906776	0.9600000000000007	——
1.00	0.998812554595766	1.0000000000000000	——

Table 4.1: Comparison of the computed roots of $P^{25}(t)$, $B^{25}_{[0,1]}(t)$, and $B^{25}_{[.25,.75]}(u)$.

the Bernstein form is much better conditioned than the power form for roots in $[\,0,1\,]$. See column 3 of Table 4.1 for the roots obtained using the Bernstein form, as constructed by multiplying out the linear factors in (4.1.1.3) using floating–point arithmetic. (Note that *transformations* between the power and Bernstein forms should be avoided [DD88, Far91b]; algorithms of interest may readily be formulated directly in the Bernstein basis [FR88] in lieu of the power basis.)

We now illustrate the effect of "*a priori* subdivision" on the root accuracy of (4.1.1.3) by constructing the Bernstein representation of this polynomial on a subinterval $[\,a,b\,] \subset [\,0,1\,]$ as follows. First, we define the change of variables

$$u = \frac{t-a}{b-a} \tag{4.1.1.4}$$

that maps $t \in [\,a,b\,]$ to $u \in [\,0,1\,]$. We then represent each of the linear factors $k(1-t) + (k-n)t$ in (4.1.1.3) in terms of u, i.e., we re–write these factors in the equivalent form

$$(k-na)(1-u) + (k-nb)u\,, \tag{4.1.1.5}$$

and then multiply them together to obtain

$$B^n_{[a,b]}(u) \;=\; \prod_{k=1}^{n} \left[\,(k-na)(1-u) + (k-nb)u\,\right] \;=\; \sum_{k=0}^{n} \tilde{b}_k \binom{n}{k} (1-u)^{n-k} u^k\,. \tag{4.1.1.6}$$

For the case $[\,a,b\,] = [0.25, 0.75]$ we find that the coefficients $\tilde{b}_k$ in (4.1.1.6) are of *much smaller magnitude* than those b_k for the representation (4.1.1.3) on the full interval $[\,0,1\,]$. Furthermore, as shown in column 4 of Table 4.1, when using this representation the twelve roots that actually lie on the interval $t \in [0.25, 0.75]$ *are computed to* ~ 13 *accurate digits* — almost twice as many as were obtained using the representation $B^{25}_{[0,1]}(t)$.

For higher degree polynomials, this experiment yields even more impressive results. For example, $n = 37$ is the highest degree for which the roots of $B^n_{[0,1]}(t)$ can be computed to at least *one* digit of accuracy in double–precision floating point: for $n = 38$, some of the computed roots become complex. Yet tests indicate that *a priori* subdivision allows us to

approximate the roots of Wilkinson polynomials of very high degree to full double precision accuracy if the interval width $b-a$ is sufficiently small. For the case $n = 1000$, for example, the 101 roots within the interval $[a,b] = [0.45, 0.55]$ can be computed to the full ~ 17 digits of accuracy if *a priori* subdivision is used when constructing the polynomial.

It should be emphasized that the dramatic improvement seen in Table 4.1 would *not* have occurred with "*a posteriori* subdivision" — i.e., if we had adopted the approach of first constructing the representation (4.1.1.3) on $[0,1]$ and then explicitly subdividing this representation to $[a,b]$ using (say) the de Casteljau algorithm. With such an approach, the quality of the computed roots is found to be comparable to, or even slightly worse, than those given in column 3 of Table 4.1. Using *exact* arithmetic, of course, one would find that the "a priori" and "a posteriori" subdivision paradigms yield *identical* results.

4.1.2 Relative Errors in the Construction Process

An empirical explanation for the improvements obtained using a priori subdivision can be given as follows. The construction of the Bernstein form of Wilkinson's polynomial was performed for degrees $10 \le n \le 100$, and intervals $[a,b]$ of width 1, 0.5, 0.25, 0.125, and 0.0625, in each case using both *double* and *quadruple* precision floating–point arithmetic. The *relative* coefficient errors incurred by the construction process were then estimated by regarding the outcome of the quadruple–precision calculations as "exact" reference values. Characteristic measures of these errors were formulated in the 1, 2, and ∞ norms (see [§ 4.2] below), and were found not to depend significantly on the chosen norm.

For a given degree, the relative errors of the constructed coefficients *showed no systematic dependence on the interval width*, and in all cases were quite small and *within a few orders of magnitude of each other* (typically 10^{-16} to 10^{-14}). Further, any tendency for the errors to increase with n was quite mild, and barely discernible within the ~ 2 orders–of–magnitude randomness. These results are consistent with the commonly–accepted notion that each

floating–point operation incurs only a small relative error [Wil60], bounded by the "machine unit" η, and the fact that the number of operations required to construct the polynomial is *independent* of $[\,a,b\,]$ and grows only *linearly* with n.

It cannot be overemphasized, however, that η is a valid bound on the relative error *only for operands that are presumed to have exact initial floating–point representations.* If — as is the case in practice — the relative error of interest is that defined by taking arbitrary real numbers, converting them to floating–point representation, and then performing the floating–point operation under consideration, this relative error can be *arbitrarily large* compared to exact arithmetic in cases were significant *cancellation* — i.e., the subtraction of nearly identical quantities — occurs (see [§ 4.2.1] below).

Our results are thus consistent with the view that, in the *construction* process, the likelihood of significant cancellation occurring does not seem to depend in any systematic way on the chosen interval. Indeed, the ~ 2 orders–of–magnitude spread observed in the relative coefficient errors suggest occasional mild cancellations (one or two digits) arising in a fairly random fashion when constructing the coefficients.

While it seems that the coefficients may be generated on any desired interval to a more–or–less uniform level of relative accuracy, coefficient errors of a fixed relative size can have vastly different consequences for representations on large and small intervals. This is because the *root condition numbers* on larger intervals are always [FR87] — and often dramatically — bigger than on smaller intervals. The results of Table 4.1, for example, can be explained by the fact that while the constructed coefficients for the representations on $[\,0,1\,]$ and $[\,0.25,0.75\,]$ have comparable relative errors ($\sim 10^{-15}$), in the former representation the roots have a far greater sensitivity to those errors — by some 6 orders of magnitude — than in the latter. A measure of the disparity between root condition number magnitudes on various intervals may be formulated in terms of the *condition of the subdivision map*, as described in [§ 4.2.2] below.

Unfortunately, formulating detailed error analyses for the "polynomial constructions" appropriate to practical geometric algorithms is not a trivial task. It is probable that, if the likelihood of significant cancellations occurring during the construction process does not depend systematically on the chosen interval, one will always produce coefficients with comparable relative errors on different intervals. It would be difficult to establish this claim for general algorithms on a quantitative analytic basis. In [§ 4.3], however, we describe empirical evidence that convincingly supports it in an important context, namely, the curve/curve intersection problem.

4.2 Problems of Error Propagation and Amplification

We may, in principle, distinguish two kinds of processes that govern the generation or amplification of numerical errors when constructing and finding the roots of polynomials in Bernstein form on various intervals. In actual floating point implementations these processes are, of course, not entirely independent — but considering them separately facilitates a simple characterization of their consequences. We can show, furthermore, that *one* of these two processes dominates in the example described in [§ 4.1], and may alone explain the observed differences in accuracy. These processes may be illustrated by considering two modes of operation for, say, the de Casteljau algorithm:

- assuming that *exact* Bernstein coefficients on $[\,0,1\,]$ are specified initially, but the de Casteljau algorithm runs in *floating point arithmetic*, a certain *accumulation* of the rounding errors associated with each of its arithmetic steps will inevitably be incurred;

- assuming now that the de Casteljau algorithm executes *in exact arithmetic*, but there are *initial errors* in the specified Bernstein coefficients on $[\,0,1\,]$, the output of the de Casteljau algorithm will inevitably reflect a certain *magnification* of those initial errors.

The second process may be described precisely (in the worst–case sense) in terms of a *condition number* for the "subdivision map" [FN90b]; see [§ 4.2.2] below. Nominally, the first process may be analyzed by means of a *running error analysis* that yields a priori error bounds [FR87]. Unfortunately, however, this kind of analysis fails — in a manner that is not easily remedied — when significant cancellations arise due to the subtraction of near–identical quantities (as noted in [§ 4.1], it is precisely this kind of situation that gives rise to severe ill–conditioning). We discuss this problem further in [§ 4.2.1] below.

The method of *backward error analysis* attempts to quantify effects of the first process in the conceptual framework of the second: the imprecise result of each floating–point arithmetic operation is regarded as a *precise* result corresponding to perturbed operands; by working these perturbations all the way back to the initial input quantities, one formulates the cumulative effect of arithmetic errors in terms of the sensitivity of the solution to perturbed input values. We will return to this in [§ 4.2.3].

4.2.1 Error Analysis of de Casteljau Algorithm

In evaluating polynomials that have been "constructed" in floating–point arithmetic, one must account for both the *propagation* of initial errors in the computed coefficients, and the *generation* — and subsequent propagation — of errors in each step of the evaluation algorithm. Similar methods of error analysis may, in principle, be applied to both the construction and evaluation algorithms. Unfortunately, however, such methods often fail to account in a tractable manner for the potentially unbounded growth of relative error (as compared to exact arithmetic) that may arise in exceptional cases of severe cancellation.

Since the "construction" algorithm will depend in detail on the geometric operation under consideration, we illustrate this problem in the context of polynomial evaluation (using the de Casteljau algorithm) only. Let $b_0, \ldots, b_n$ be the Bernstein coefficients of a degree–n polynomial $P(t)$ on $[\,0, 1\,]$ and let $s \in (0, 1)$ be a point at which we wish to evaluate

— and "split" — the polynomial. Upon setting $b_k^{(0)} = b_k$ for $k = 0, \ldots, n$, the de Casteljau algorithm consists of the iterated sequence of linear interpolations:

$$b_k^{(r)} \;=\; (1-s)\, b_{k-1}^{(r-1)} \;+\; s\, b_k^{(r-1)} \qquad k = r, r+1, \ldots, n \tag{4.2.1.7}$$

for $r = 1, 2, \ldots, n$. The quantities $b_k^{(r)}$ form a triangular array, and the values

$$b_0^{(0)},\, b_1^{(1)},\, \ldots,\, b_n^{(n)} \quad \text{and} \quad b_n^{(n)},\, b_n^{(n-1)},\, \ldots,\, b_n^{(0)}$$

on the left– and right–hand sides of this array are the Bernstein coefficients for $P(t)$ on the two subintervals $[\,0, s\,]$ and $[\,s, 1\,]$. In particular, $b_n^{(n)}$ represents the *value* $P(s)$ of the polynomial at the split point.

The (forward) error analysis of an algorithm such as (4.2.1.7) is usually based [Wil60] on the formula

$$(1-\eta)\, x * y \;\le\; \text{float}(x * y) \;\le\; (1+\eta)\, x * y\,, \tag{4.2.1.8}$$

where $\text{float}(x * y)$ denotes the result of a floating–point arithmetic operation "$*$" (i.e., $+$, $-$, $\times$, or $\div$), $x * y$ is the exact–arithmetic result, and $\eta = \frac{1}{2} b^{-(d-1)}$ represents the *machine unit* for floating–point numbers having a mantissa of d digits in base b (we assume rounding, rather than truncation).

Basically, equation (4.2.1.8) says that relative error incurred in each floating–point arithmetic operation is bounded by a very small quantity η (for typical double–precision arithmetic, with $b = 2$ and $d = 53$, we have $\eta \approx 10^{-16}$). Great caution must be urged in attaching a "practical" significance to this statement, however. It is implicit in (4.2.1.8) that the operands x and y are numbers that have exact floating–point representations *ab initio*. This is hardly ever true: in practice, x and y usually represent *rounded* approximations to arbitrary real numbers X and Y that may be input values or results of intermediate calculations. We denote by $x = \text{float}(X)$ and $y = \text{float}(Y)$ this process of rounding, which itself incurs only small fractional errors:

$$(1-\eta)\, X \;\le\; x = \text{float}(X) \;\le\; (1+\eta)\, X\,,$$

$$(1-\eta)\,Y \;\le\; y = \text{float}(Y) \;\le\; (1+\eta)\,Y\,. \tag{4.2.1.9}$$

For practical purposes, of course, the error of interest incurred by the operation (4.2.1.8) is *the error relative to the result of an exact–arithmetic operation on arbitrary real numbers* X *and* Y, i.e., it is the quantity

$$\text{relative error} \;=\; \frac{|\text{float}(x * y) \;-\; X * Y|}{|X * Y|}\,. \tag{4.2.1.10}$$

In most instances, this quantity is — in agreement with intuition — no more than a few times η, and the model (4.2.1.8) thus provides a fairly reliable basis for analyzing error propagation and estimating the deviation of floating–point calculations from exact–arithmetic results. There are important circumstances, however, in which the quantity (4.2.1.10) can exceed η by many orders of magnitude, and the model (4.2.1.8) fails dramatically in terms of monitoring the discrepancy between floating–point and exact–arithmetic results.

Such instances correspond to *the subtraction of like–signed floating–point numbers* x *and* y *that have identical exponents and agree in several leading digits of their mantissas* (or, equivalently, the addition of such quantities of unlike sign). In such situations, cancellation of the identical leading digits of x and y occurs, and the quantity $x * y$ therefore requires no rounding for machine representation: $\text{float}(x * y) \equiv x * y$. The problem is thus not one of *arithmetic error* in the actual subtraction, but rather *magnification of "initial" errors incurred in the floating–point conversions* $x = \text{float}(X)$, $y = \text{float}(Y)$.

To illustrate this, we use (4.2.1.9) to write $x = (1+\delta)X$ and $y = (1+\epsilon)Y$, where δ and ϵ are random numbers in the range $-\eta \le \delta, \epsilon \le +\eta$. We further suppose that X and Y agree in r of their leading binary digits. Noting again that $\text{float}(x * y) \equiv x * y$ in such a case, expression (4.2.1.10) becomes

$$\text{relative error} \;=\; \frac{|(1+\delta)X - (1+\epsilon)Y \;-\; (X-Y)|}{|X-Y|} \;=\; \frac{|\delta X - \epsilon Y|}{|X-Y|}\,.$$

Now the numerator of the right–hand side attains its maximum value, namely $\eta\,(|X|+|Y|)$, when δ and ϵ are of opposite sign and magnitude η. Furthermore, if X and Y have mantissas

that agree in r leading bits, we have

$$|X| + |Y| \approx 2|X| \quad \text{and} \quad |X - Y| \approx 2^{-r}|X| \,,$$

so the bound on the error, *relative to the exact–arithmetic result*, is approximately

$$\text{relative error} \lesssim 2^{r+1}\,\eta \,. \tag{4.2.1.11}$$

This bound can clearly become arbitrarily large as $r \to \infty$ (note that r is the number of bits of agreement of the exact real numbers X and Y, and is not bounded by d, the number of bits of the mantissa in the floating–point number system). We emphasize again the nature of the phenomenon that gives rise to the error (4.2.1.11): it is not at all an "arithmetic error" — it is rather a *magnification* of the small relative errors incurred by the conversions $x = \text{float}(X)$ and $y = \text{float}(Y)$ that arises when $|X - Y| \ll |X|, |Y|$.

The inability of the simple model (4.2.1.8) to accommodate the large relative errors (4.2.1.11) due to cancellation effects is a serious shortcoming. Moreover, this defect cannot be easily "fixed" in a manner that is germane to a general–purpose analysis (rather than case–by–case inspections) — the amount of cancellation, which determines the error–amplification factor in (4.2.1.11), is evidently dependent on the detailed numerical values of X and Y. Note also that error amplification due to cancellation effects need not be confined to individual arithmetic operations; the calculation (4.1.0.2) is an example of a sum of many terms where the final result is extremely susceptible to such effects.

Notwithstanding this serious defect, the model (4.2.1.8) is widely used, and can yield reasonable error estimates (compared to exact arithmetic) in cases where one can be confident that cancellation effects will not arise. By applying this model to the algorithm (4.2.1.7), for example, one obtains the (rather loose) bound [FR87]:

$$|\delta P| \;\leq\; 2n \max_k(|b_k|)\,\eta$$

on the absolute error in $P(s) = b_n^{(n)}$ due to arithmetic round–off. One can be confident of this bound in, for example, cases where $s \in [\,0, 1\,]$ and the initial coefficients $b_0, \ldots, b_n$

are all of like sign.[1] The important point to note is that, in such cases, the arithmetic error is well contained — it grows only *linearly* with n and does not "explode" as we consider polynomials of ever–higher degree. This is in sharp contrast to the cancellation phenomenon described above, and the related matter of *amplification of initial errors* in $b_0, \ldots, b_n$ as described in [§ 4.2.2] below.

Another point worth mentioning with regard to (4.2.1.8) is that one should be wary of naively using this model on modern RISC machines, which may have, for example, a hardware FMA (i.e., floating–point multiply–and–add) instruction that allows expressions such as $(x \times y) + z$ to be evaluated in a single cycle and with an error lower than ordinary sequential $\times$ and $+$ operations would incur. The ability to utilize such machine instructions can be algorithm– and compiler–dependent [Bel90].

We may summarize as follows. In the absence of cancellation effects, the accumulation of arithmetic error in the execution of an algorithm (as compared to exact–arithmetic results) is well described by the formula (4.2.1.8), which indicates very small relative errors in each step and consequently fairly mild final errors in all but very lengthy calculations. When cancellation effects occur, however, the model (4.2.1.8) fails dramatically to provide a reliable indication of the deviation from exact–arithmetic results, and cannot be easily amended to do so. In such cases, the errors that arise are not really *arithmetic* errors, but rather *magnifications* of pre–existing errors. Although we have emphasized the de Casteljau algorithm, similar considerations apply to the "polynomial construction" algorithms corresponding to various geometric operations of interest.

4.2.2 Condition of the Subdivision Map

We now turn to the second process mentioned at the beginning of [§ 4.2]. Given the coefficients $\mathbf{b} = (b_0, \ldots, b_n)^T$ of a degree–n polynomial in the Bernstein basis on $t \in [\,0, 1\,]$

[1] A rather uninteresting case in the present context, since $P(t)$ would evidently have no real roots on $t \in [\,0, 1\,]$.

we may obtain the corresponding coefficients $\hat{\mathbf{b}} = (\hat{b}_0, \ldots, \hat{b}_n)^T$ in the Bernstein basis on any subinterval $t \in [\, a, b\,]$ thereof by multiplying the former by the appropriate $(n+1) \times (n+1)$ *subdivision matrix*:

$$\hat{\mathbf{b}} \;=\; \mathbf{M}\,\mathbf{b}\,. \tag{4.2.2.12}$$

Note that multiplication by $\mathbf{M}$, whose elements are given by [FN90b]:

$$M_{jk} \;=\; \sum_{i\,=\,\max(0,j+k-n)}^{\min(j,k)} \binom{n-j}{k-i}(1-a)^{n-j-k+i}a^{k-i}\binom{j}{i}(1-b)^{j-i}b^i \qquad \text{for } \; 0 \le j,k \le n\,,$$

is equivalent to two applications of the de Casteljau algorithm. We are concerned here with the *condition* or "inherent stability" of the linear map defined by (4.2.2.12) — i.e., a bound C on the amount by which (random) errors of fractional magnitude ϵ in the input coefficients $\mathbf{b}$ are amplified to give the fractional errors $\hat{\epsilon}$ of the output coefficients $\hat{\mathbf{b}}$, so that $\hat{\epsilon} \le C\epsilon$. This "error amplification" is intrinsic to the analytic relation between $\hat{\mathbf{b}}$ and $\mathbf{b}$ — it would arise even if the multiplication (4.2.2.12) were performed in *exact arithmetic*.

Introducing the vector and subordinate matrix p–norms defined by

$$\|\mathbf{b}\|_p \;=\; \left[\,\sum_{k=0}^{n} |b_k|^p\right]^{1/p} \qquad \text{and} \qquad \|\mathbf{M}\|_p \;=\; \sup_{\mathbf{b}\neq\mathbf{0}} \frac{\|\mathbf{M}\mathbf{b}\|_p}{\|\mathbf{b}\|_p}\,,$$

and introducing p–norm fractional error measures for $\mathbf{b}$ and $\hat{\mathbf{b}}$ as

$$\epsilon_p \;=\; \frac{\|\delta\mathbf{b}\|_p}{\|\mathbf{b}\|_p} \qquad \text{and} \qquad \hat{\epsilon}_p \;=\; \frac{\|\delta\hat{\mathbf{b}}\|_p}{\|\hat{\mathbf{b}}\|_p}\,,$$

where $\delta\mathbf{b} = (\delta b_0, \ldots, \delta b_n)^T$ and $\delta\hat{\mathbf{b}} = (\delta\hat{b}_0, \ldots, \delta\hat{b}_n)^T$ represent the *individual* coefficient errors, it is readily verified [Ste73] that

$$\hat{\epsilon}_p \;\le\; C_p(\mathbf{M})\,\epsilon_p \qquad \text{where } \; C_p(\mathbf{M}) \;=\; \|\mathbf{M}\|_p\|\mathbf{M}^{-1}\|_p\,. \tag{4.2.2.13}$$

The quantity $C_p(\mathbf{M})$ is the p–norm *condition number* of the subdivision matrix. Note that the error bound (4.2.2.13) is *sharp*, i.e., there exist perturbations $\delta\mathbf{b} = (\delta b_0, \ldots, \delta b_n)^T$ of the given coefficients such that $\hat{\epsilon}_p = C_p(\mathbf{M})\,\epsilon_p$. In the case $p = \infty$, where we define

$\|\mathbf{b}\|_\infty = \max_k(|b_k|)$, the condition number defined by (4.2.2.13) admits a simple closed–form expression [FN90b]:

$$C_\infty(\mathbf{M}) = \left[\frac{2\max(m, 1-m)}{b-a} \right]^n ,$$

where $m = (a+b)/2$ is the midpoint of the subinterval $[\,a, b\,]$. Analogous expressions for other norms are not known, although it is fairly easy to compute numerical values in two cases: $C_2(\mathbf{M}) = \sqrt{\lambda_{\max}/\lambda_{\min}}$, where $\lambda_{\max}$ and $\lambda_{\min}$ denote the largest and smallest eigenvalues of the symmetric matrix $\mathbf{M}^T\mathbf{M}$, while $C_1(\mathbf{M})$ is the product of the greatest column–sums of absolute values of the elements of $\mathbf{M}$ and $\mathbf{M}^{-1}$.

Table 4.2 lists computed values for the logarithm of the subdivision matrix condition number for the case $[\,a, b\,] = [\,0.25, 0.75\,]$ and degrees $1 \le n \le 25$. These values indicate, approximately, the number of additional decimal places of accuracy that may be lost in "magnifying" (i.e., subdividing) the given representation on $[\,0, 1\,]$ to that on $[\,0.25, 0.75\,]$. It is seen that the magnitude of the condition number does not depend strongly on the chosen norm p, and in each case exhibits a roughly *exponential* growth with the polynomial degree n (a unit increase in degree incurs roughly a doubling of the condition number).

It is worthwhile noting that the amplification of the relative errors ϵ_p in the coefficients on $[\,0, 1\,]$ by a factor of up to $C_p(\mathbf{M})$ to give the relative errors $\hat{\epsilon}_p$ on $[\,a, b\,]$ is mostly due, in the present instance, to a *decrease* in $\|\hat{\mathbf{b}}\|_p$ compared to $\|\mathbf{b}\|_p$, rather than an *increase* in $\|\delta\hat{\mathbf{b}}\|_p$ relative to $\|\delta\mathbf{b}\|_p$. From the relations $\delta\hat{\mathbf{b}} = \mathbf{M}\,\delta\mathbf{b}$ and $\mathbf{b} = \mathbf{M}^{-1}\,\hat{\mathbf{b}}$, and the definition of the matrix norm, we note that

$$\|\delta\hat{\mathbf{b}}\|_p \le \|\mathbf{M}\|_p \|\delta\mathbf{b}\|_p \qquad \text{and} \qquad \|\hat{\mathbf{b}}\|_p \ge \frac{\|\mathbf{b}\|_p}{\|\mathbf{M}^{-1}\|_p} ,$$

and the subdivision matrix $\mathbf{M}$ that we are interested has the property $\|\mathbf{M}\|_p \ll \|\mathbf{M}^{-1}\|_p$ for large n or small $[\,a, b\,]$ (in fact, $\|\mathbf{M}\|_p \equiv 1$ for the case $p = \infty$ [FN90b]). In other words, *relative* coefficient errors increase with subdivision because the coefficient magnitudes decrease while the *absolute* errors remain relatively constant.

n	$\log_{10} C_1$	$\log_{10} C_2$	$\log_{10} C_\infty$
1	0.301	0.301	0.301
2	0.879	0.633	0.602
3	1.180	0.963	0.903
4	1.605	1.288	1.204
5	1.906	1.609	1.505
6	2.276	1.926	1.806
7	2.577	2.241	2.107
8	2.921	2.553	2.408
9	3.222	2.864	2.709
10	3.551	3.174	3.010
11	3.852	3.483	3.311
12	4.171	3.790	3.612
13	4.472	4.098	3.913
14	4.786	4.403	4.214
15	5.087	4.704	4.515
16	5.397	5.018	4.816
17	5.698	5.370	5.118
18	6.005	5.672	5.419
19	6.306	5.975	5.720
20	6.611	6.278	6.021
21	6.912	6.580	6.322
22	7.217	6.883	6.623
23	7.518	7.185	6.924
24	7.821	7.487	7.225
25	8.122	7.741	7.526

Table 4.2: Condition numbers of subdivision matrices in various norms.

The condition number $C_p(\mathbf{M})$ was formulated above as a bound on the amount by which initial fractional errors in the Bernstein coefficients on $[\,0,1\,]$ are amplified when the coefficients on $[\,a,b\,]$ are determined from them. Another characterization is useful in interpreting the empirical results described in [§ 4.1]: if one may choose between Bernstein coefficients on $[\,0,1\,]$ or $[\,a,b\,]$ of *comparable relative accuracy*, then the former choice leads to errors in the roots/values of the polynomial between a and b that are up to $\sim C_p(\mathbf{M})$ times larger than the latter. This may be explained intuitively as follows:

It was shown in [FR87] that, for an arbitrary polynomial, the root condition numbers in the Bernstein basis on $[\,a,b\,]$ are smaller than those in the basis on $[\,0,1\,]$. How much smaller depends, of course, on the particular polynomial and root under consideration. But we now argue that, for *all* polynomials, $C_p(\mathbf{M})$ represents a fair measure of the largest factor by which root condition numbers on $[\,0,1\,]$ exceed those on $[\,a,b\,]$.

Suppose a root r has condition number k in the basis on $[\,0,1\,]$ so that an (infinitesimal) fractional error ϵ in the Bernstein coefficients on that interval induces an uncertainty in the root satisfying $|\delta r| \leq k\,\epsilon$. If we now subdivide down to $[\,a,b\,]$ *in exact arithmetic*, we know the root r must have a smaller condition number, $\hat{k}$ say, in the basis on this subinterval. The exact subdivision, however, clearly cannot alter the uncertainty in r, and hence the diminution of the root condition number must be accompanied by a corresponding increase of the fractional coefficient errors to, say, $\hat{\epsilon}$. Setting $|\delta r| \leq k\,\epsilon = \hat{k}\,\hat{\epsilon}$, we see from (4.2.2.13) that the root condition numbers on $[\,0,1\,]$ must be larger than those on $[\,a,b\,]$ by a factor of up to

$$k/\hat{k} \;=\; \hat{\epsilon}/\epsilon \;=\; C_p(\mathbf{M})\,.$$

We spoke only of roots above, but the same considerations hold for the *values* of a polynomial between a and b. It follows as a corollary of our argument that one must typically know the Bernstein coefficients on $[\,0,1\,]$ to a precision some $C_p(\mathbf{M})$ times greater than that of those on $[\,a,b\,]$ so as to guarantee root/value computations based on the former

of accuracy comparable to that afforded by the latter. Note from Table 4.2, for example, that $C_p(\mathbf{M}) \gtrsim 10^6$ for $n \geq 20$. Assuming that the relative errors incurred in "constructing" the Bernstein form of Wilkinson's polynomial are of comparable magnitude when $[\,a, b\,]$ is chosen to be either $[\,0, 1\,]$ or $[\,0.25, 0.75\,]$, the phenomenon described above is *alone* sufficient to explain the fact that the roots computed using the former representation have ~ 6 fewer accurate decimal digits than those using the latter.

4.2.3 Backward Error Analysis

In [§ 4.2.1] we have emphasized the difficulty — due mainly to cancellation effects — of formulating reliable measures for the accuracy of computed results, relative to corresponding exact–arithmetic calculations, based on models for the (forward) propagation of floating–point arithmetic errors. Although more subtle, and often more difficult to carry out in practice, the method of *backward error analysis* [Wil60, Wil63] — coupled with condition number estimates for the problem at hand — can largely bypass this difficulty.

The basic idea underlying backward error analysis is to note that the result of floating–point arithmetic operations on operands x and y will satisfy

$$\begin{aligned} \text{float}(x \times y) &\equiv (1+\delta)\, xy\,, \\ \text{float}(x \div y) &\equiv (1+\delta)\, x/y\,, \\ \text{float}(x \pm y) &\equiv (1+\delta)x \pm (1+\epsilon)y\,, \end{aligned} \tag{4.2.3.14}$$

for some $\delta, \epsilon \in [\,-\eta, +\eta\,]$. By contrast with (4.2.1.8), the formulations (4.2.3.14) should *not* be interpreted as furnishing bounds for the error incurred in each floating–point arithmetic operation. Instead, they state that the outcome of such operations will be identical to the results of *exact–arithmetic* operations on perturbed operands. For multiplication/division, we associate a perturbation of relative magnitude $\leq \eta$ with only *one* of the operands; for addition/subtraction, we must allow for such a perturbation in *both* operands [Wil63].

Applying this notion to each arithmetic step of an algorithm, one can say that the final

result obtained by a floating–point computation is the *exact* result for some "neighboring" problem, corresponding to perturbed input values. The key requirement is thus to propagate the individual perturbations of operands in each step backward so as to obtain the appropriate overall perturbations of the input. To complete the analysis, one must also have a measure of the sensitivity of the final answer to input perturbations of a given magnitude, i.e., one must be able to compute the *condition number* of the problem at hand.

Thus, the method does not attempt a detailed step–by–step comparison of corresponding floating–point and exact–arithmetic calculations (which was the cause of the difficulties described in [§ 4.2.1]). Rather, it says that the floating–point result *is* exact for some "nearby" problem (one having slightly different input data). Whether this problem is sufficiently nearby to produce an accurate answer will depend, of course, on whether or not neighboring problems have neighboring results, i.e., on the *condition* of the problem.

To illustrate the use of the backward–error–analysis method, consider its application to the de Casteljau algorithm (4.2.1.7). Assuming, for simplicity, that both $1-s$ and s have exact floating–point representations, the evaluation of (4.2.1.7) in floating point using the rules (4.2.3.14) becomes

$$\begin{aligned} b_k^{(r)} &= \mathrm{float}\Big[\,\mathrm{float}\Big((1-s)\times b_{k-1}^{(r-1)}\Big) \;+\; \mathrm{float}\Big(s\times b_k^{(r-1)}\Big)\,\Big] \\ &\equiv (1+\delta_k^{(r)})(1+\mu_k^{(r)})\,(1-s)\,b_{k-1}^{(r-1)} \;+\; (1+\epsilon_k^{(r)})(1+\nu_k^{(r)})\,s\,b_k^{(r-1)}\,. \end{aligned} \qquad (4.2.3.15)$$

Here $\mu_k^{(r)}$ and $\nu_k^{(r)}$ are perturbations associated with the two multiplications, and $\delta_k^{(r)}$, $\epsilon_k^{(r)}$ are perturbations associated with the single addition; these quantities are all of magnitude $\leq \eta$. We emphasize again the interpretation that is to be attached to (4.2.3.15): it represents an *exact* execution of the de Casteljau step (4.2.1.7), but on the perturbed operands

$$(1+\delta_k^{(r)})(1+\mu_k^{(r)})\,b_{k-1}^{(r-1)} \quad \text{and} \quad (1+\epsilon_k^{(r)})(1+\nu_k^{(r)})\,b_k^{(r-1)}\,,$$

rather than the nominal values $b_{k-1}^{(r-1)}$ and $b_k^{(r-1)}$.

Suppose we are interested only in the final computed value $b_n^{(n)} = \mathrm{float}(P(s))$ of the

polynomial, i.e., the value produced by a floating–point computation. By carrying through the steps (4.2.3.15) it is possible to express this value in the form

$$\text{float}(P(s)) \;=\; \sum_{j=0}^{n} (1+e_j) b_j \binom{n}{j} (1-s)^{n-j} s^j \,, \tag{4.2.3.16}$$

namely, the computed quantity is the *exact* value of a polynomial with "perturbed" coefficients $(1+e_0)b_0, \ldots, (1+e_n)b_n$, rather than the nominal coefficients $b_0, \ldots, b_n$. It is not difficult — only rather tedious — to derive closed–form expressions for the overall coefficient perturbations $e_0, \ldots, e_n$ in terms of the individual arithmetic–step perturbations

$$\mu_k^{(r)}, \nu_k^{(r)} \quad \text{and} \quad \delta_k^{(r)}, \epsilon_k^{(r)} \qquad \text{for } k = r, \ldots, n \text{ and } r = 1, \ldots, n \,.$$

Such expressions would, in any case, require simplification to be of practical value. This can be achieved, without formally writing down the expressions, as follows:

Upon execution of (4.2.3.15) for $k = r, \ldots, n$ and $r = 1, \ldots, n$, the j–th term in (4.2.3.16) is evidently the (exact) sum of $\binom{n}{j}$ perturbed quantities, each of which may be traced back through n floating–point multiplications and n floating–point additions to the original coefficient b_j. By allowing also for the need to round the initial coefficients to floating–point format, it is seen that the perturbation factors in (4.2.3.16) have the form

$$1 + e_j \;=\; \langle 2n+1 \rangle$$

where, following Stewart [Ste73], we have introduced the notation

$$\langle m \rangle \;=\; \prod_{k=1}^{m} 1 + \lambda_k \,, \tag{4.2.3.17}$$

the λ_k denoting appropriate choices and indices for the individual–step perturbations $\mu_k^{(r)}$, $\nu_k^{(r)}$ and $\delta_k^{(r)}$, $\epsilon_k^{(r)}$ (each bounded by η in magnitude). We are thus interested in establishing bounds on the quantity (4.2.3.17). If $\eta \ll 1$, we clearly have the *approximate* bounds

$$1 - m\eta \;\lesssim\; \langle m \rangle \;\lesssim\; 1 + m\eta \,.$$

If we assume (as is reasonable in most practical circumstances) that $m\eta < 0.1$, more rigorous bounds may be derived [DB75, Wil63] as follows. Noting that

$$(1+\eta)^m \;<\; \exp(m\eta)$$

(since each term in the binomial expansion of the left–hand side is less than or equal to the corresponding term in the Taylor series for the right–hand side), we have

$$(1+\eta)^m - 1 \;<\; m\eta\left[1 + \frac{m\eta}{2!} + \frac{(m\eta)^2}{3!} + \cdots\right] \;<\; m\eta\left[1 + \frac{m\eta}{2} + \left(\frac{m\eta}{2}\right)^2 + \cdots\right],$$

where the second step follows from the fact that $r! \geq 2^{r-1}$ for all $r \geq 1$. The infinite sum in the final expression is just $(1-\frac{1}{2}m\eta)^{-1}$ and thus $(1+\eta)^m - 1 < m\eta\,(1-\frac{1}{2}m\eta)^{-1} \lesssim 1.053\,m\eta$ when $m\eta < 0.1$. Conventionally [Wil63], we write

$$1 - 1.06m\eta \;<\; \langle m \rangle \;<\; 1 + 1.06m\eta\,.$$

We are now ready to state the results of our backward error analysis for the de Casteljau algorithm. Let $b_0, \ldots, b_n$ be the Bernstein coefficients of a degree–n polynomial, and let the value $s \in [\,0, 1\,]$ be such that both $1 - s$ and s have exact floating point representations. Then the outcome $b_n^{(n)} = \mathrm{float}(P(s))$ of the algorithm (4.2.1.7) running in floating point is the *exact* value of a polynomial with Bernstein coefficients $(1+e_0)b_0, \ldots, (1+e_n)b_n$, where the perturbations e_j satisfy

$$|e_j| \;<\; 1.06\,(2n+1)\,\eta \qquad \text{for } \; j = 0, \ldots, n\,. \tag{4.2.3.18}$$

Note that these perturbations are of *uniform magnitude*, i.e., independent of the index j. Thus, evaluation of a polynomial in Bernstein form using de Casteljau's algorithm in floating–point arithmetic yields the *exact* value for some "nearby" polynomial, corresponding to a uniform (random) perturbation of the coefficients of relative magnitude $\epsilon = 1.06(2n+1)\eta$ (compare with the evaluation of a polynomial in power form by Horner's method, for which larger perturbations must be associated with higher–order coefficients).

In order to complete the argument, we need to specify what effect a relative perturbation ϵ in the coefficients $b_0, \ldots, b_n$ will have on the values/roots of the polynomial $P(t)$ by formulating the appropriate *condition numbers* C_v (for values) and C_r (for roots). In the present case, we have [FR87]:

$$|\delta P| \leq C_v \epsilon \quad \text{where} \quad C_v = \sum_{j=0}^{n} \left| b_j \binom{n}{j} (1-t)^{n-j} t^j \right| \tag{4.2.3.19}$$

for the bound on the error in the value at any point t, and

$$|\delta r| \leq C_r \epsilon \quad \text{where} \quad C_r = \frac{1}{|P'(r)|} \sum_{j=0}^{n} \left| b_j \binom{n}{j} (1-r)^{n-j} r^j \right| \tag{4.2.3.20}$$

for the bound on the displacement of a root r, i.e., $P(r) = 0$; the latter bound holds only for infinitesimal ϵ.

With $\epsilon = 1.06(2n+1)\eta$, expressions (4.2.3.19) and (4.2.3.20) describe errors in computed values and roots that are due to *floating–point arithmetic errors incurred in using the de Casteljau algorithm* (4.2.1.7) *to evaluate the polynomial.* The backward–error–analysis and condition–number approach, as described above, fully accommodates the possibility of large error amplifications due to cancellation effects.

4.3 Application to Curve/Curve Intersections

The mode of "constructing" polynomials used in the examples of [§ 4.1] is admittedly rather artificial — *knowing* the roots of a polynomial, we multiplied out the linear factors corresponding to each of those roots to obtain the polynomial coefficients in various bases. We will now show that the "pre–conditioning" strategy of using representations that have been subdivided (even in floating point arithmetic) *before* performing the polynomial construction, rather than constructing the polynomial on $[\,0, 1\,]$ and then subdividing, can also yield significant improvements in accuracy in the context of more "realistic" calculations.

Specifically, we choose the context of computing the intersection points of two polynomial Bézier curves, by means of the "implicitization" algorithm [SP86], to illustrate this.

Suppose

$$\mathbf{r}_1(t) = \{x_1(t), y_1(t)\} \quad \text{and} \quad \mathbf{r}_2(t) = \{x_2(t), y_2(t)\}$$

are the curves in question, of degree n_1 and n_2. The Bézier control points $\mathbf{p}_{1,k}$ for $k = 0, \ldots, n_1$ and $\mathbf{p}_{2,k}$ for $k = 0, \ldots, n_2$ of the two curves (defined on $t \in [\,0, 1\,]$) are the input data; the output data are the parameter values $t_1, \ldots, t_N \in [\,a, b\,]$, where $N \leq n_1 n_2$, on a prescribed subsegment of (say) $\mathbf{r}_1(t)$ that identify its intersections with $\mathbf{r}_2(t)$.

The "intersection polynomial" $P(t)$, of degree $n_1 n_2$, is to be constructed by substituting the parametric equations $x = x_1(t)$, $y = y_1(t)$ of the first curve into the implicit equation $f_2(x, y) = 0$ of the second: $P(t) = f_2(x_1(t), y_1(t))$. The real roots of this polynomial correspond to parameter values on $\mathbf{r}_1(t)$ where the two curves intersect. The bivariate polynomial $f_2(x, y)$ describing the curve $\mathbf{r}_2(t)$ implicitly may be formulated as an $n_2 \times n_2$ determinant, whose entries are the linear forms

$$L_{ij}(x, y) = a_{ij}x + b_{ij}y + c_{ij} = \sum_{\substack{k \leq \min(i,j) \\ k+l = i+j+1}} \binom{n_2}{k} \binom{n_2}{l} (\mathbf{p}_{2,k} - \mathbf{r}) \times (\mathbf{p}_{2,l} - \mathbf{r}) \qquad (4.3.0.21)$$

for $0 \leq i, j \leq n_2 - 1$, where we write $\mathbf{r} = (x, y)$. The curve intersection problem thus breaks down naturally into two stages: (i) construction of $P(t)$ on a prescribed interval $[\,a, b\,]$ by substituting the Bézier form of $\mathbf{r}_1(t)$ on $[\,a, b\,]$ for $\mathbf{r}$ in the determinant defined by (4.3.0.21) and expanding it out; and (ii) finding the roots of $P(t)$ on the interval $[\,a, b\,]$ based on this constructed representation.

A formal error analysis of the curve intersection problem, along the lines described in [§ 4.2.3], would comprise the following steps:

- Perform a *backward error analysis* for the construction of the intersection polynomial $P(t)$ on $[\,a, b\,]$ in floating point arithmetic, to ascertain the relative magnitude ϵ of *initial* perturbations in the control points $\mathbf{p}_{1,k}$ and $\mathbf{p}_{2,k}$ that are equivalent to the effects of rounding errors in the floating–point expansion of the determinant defined by (4.3.0.21).

- Formulate a *condition number* C_c for the construction of $P(t)$ on $[\,a, b\,]$ as follows: if the control points $\mathbf{p}_{1,k}$ and $\mathbf{p}_{2,k}$ are subject to random perturbations $\delta\mathbf{p}_{1,k}$ and $\delta\mathbf{p}_{2,k}$ of relative magnitude ϵ in a suitable norm, then the relative magnitude of the induced errors δb_k in the Bernstein coefficients b_k of $P(t)$ on $[\,a, b\,]$ will be bounded (in the same norm) by $C_c\,\epsilon$.

- By combining the results of the preceding steps, it may be deduced that the coefficients b_k of $P(t)$, as constructed in floating point arithmetic, will have relative errors of order $C_c\,\epsilon$.

- From the condition number C_r for a root r of $P(t)$ (see expression (4.2.3.20) above), we can estimate the perturbation of the root due to floating–point errors in the *construction* process as $|\delta r| \lesssim C_r C_c\,\epsilon$. The effect of floating–point errors in *evaluating* the constructed polynomial during the root–finding process may also be incorporated using the backward error analysis described in [§ 4.2.3] — if these errors are equivalent to perturbations of relative magnitude ϵ' in the coefficients b_k, then as $|\delta r| \lesssim C_r(\epsilon' + C_c\,\epsilon)$.

Note that, if the control points $\mathbf{p}_{1,k}$ and $\mathbf{p}_{2,k}$ are specified on $[\,0, 1\,]$, their floating–point subdivision to the subinterval $[\,a, b\,]$ must be considered a part of the construction process if $P(t)$ is to be constructed on that interval (i.e., the condition and backward error analysis effects of this step must be incorporated in the quantities C_c and ϵ — these effects have already been characterized in [§ 4.2.2] and [§ 4.2.3]).

There are, unfortunately, formidable practical difficulties in carrying out the first two steps of the above program. The coefficients b_k of $P(t)$ are, in general, polynomials of degree $2\max(n_1, n_2)$ in the coordinates of the control points $\mathbf{p}_{1,k}$ and $\mathbf{p}_{2,k}$. However, writing them out explicitly as such — in order to determine their sensitivity to input perturbations $\delta\mathbf{p}_{1,k}$ and $\delta\mathbf{p}_{2,k}$ — is a cumbersome process (even if we could formulate C_c by such an approach, it would apply only to infinitesimal perturbations). By the same token, a backward error analysis for the evaluation of the determinant whose elements are given by (4.3.0.21) would

be quite involved, and dependent on the particular determinant expansion scheme adopted.

We therefore rely on empirical results to show that, for the curve intersection problem, the construction of $P(t)$ incurs relative errors in its coefficients that are fairly (in at least an average, if not worst–case, sense) independent of the choice of interval, i.e., $C_c\,\epsilon$ is not strongly dependent on $[\,a, b\,]$. Choosing smaller intervals thus gives more accurate results, on account of the smaller root condition numbers C_r associated with them.

Figure 4.1 shows a configuration of two degree–7 Bézier curves, intersecting at 49 distinct real points, that we have used as a test case. We have attempted to compute the parameter values of the intersection points by constructing the Bernstein coefficients of the intersection polynomial $P(t)$ on various intervals. Using standard double–precision arithmetic, it was found that the relative errors of the constructed coefficients (estimated by comparison with quadruple–precision calculations) were typically of order 10^{-15} and fairly independent of the chosen interval.

When the intersection polynomial was constructed on the full interval $[\,0, 1\,]$ only 15 of the 49 real roots could be computed. Performing constructions on the intervals $[\,0, \frac{1}{2}\,]$ and $[\,\frac{1}{2}, 1\,]$ instead, all 49 roots were determined to an accuracy of at least 7 significant digits (with up to 16 digits for roots near the interval endpoints). Finally, a construction on $[\,\frac{1}{6}, \frac{1}{2}\,]$ gave at least 13 accurate digits for all roots on that interval.

4.4 Concluding Remarks

For geometrical algorithms that have a natural algebraic formulation, in terms of finding the real roots of polynomials that must be constructed from given data, we have shown that the "pre–conditioning" strategy of constructing the Bernstein form on smaller intervals can lead to significant improvements in accuracy, especially for conspicuously ill–conditioned problems. Although, for reasons enumerated above, a rigorous demonstration of this claim for the general curve/curve intersection problem seems infeasible, the empirical evidence

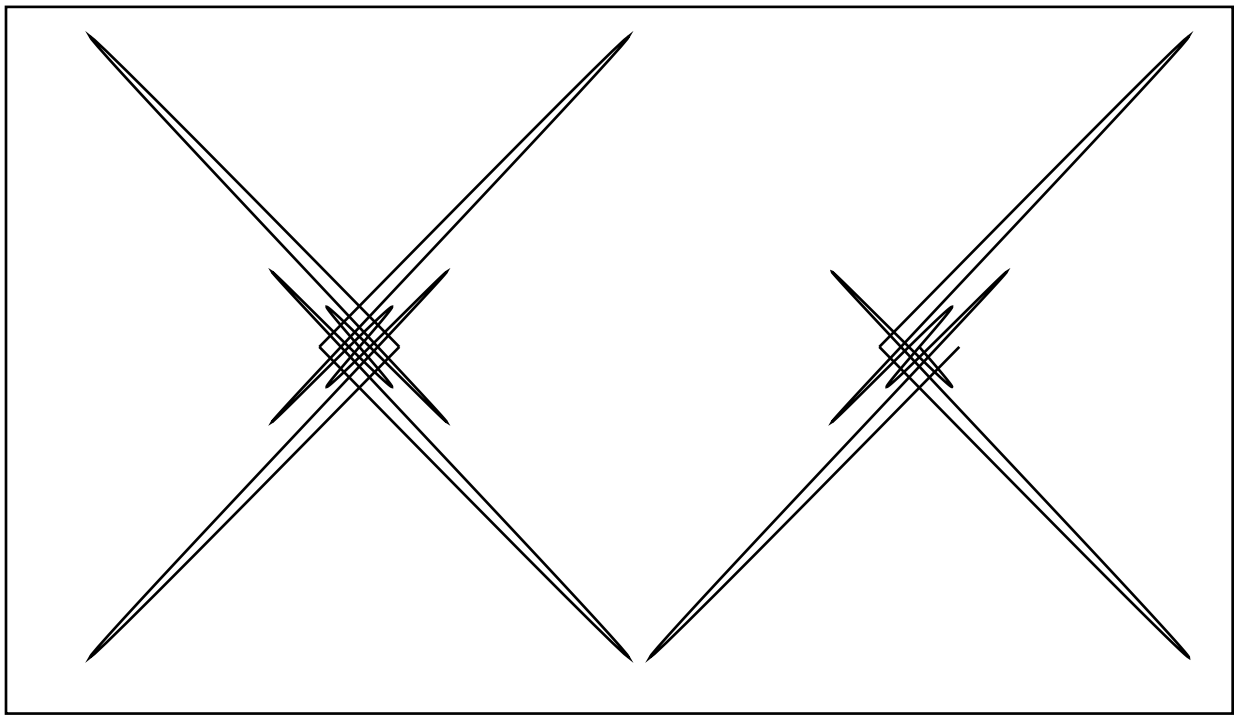

Figure 4.1: Two degree–7 Bézier curves having 49 real intersections

is compelling and suggests a simple intuitive explanation: whereas the relative accuracy with which the intersection polynomial may be "constructed" is fairly independent of the chosen interval width, Bernstein representations on smaller intervals can have significantly enhanced root stability compared to those on larger intervals.

It is known [FR87] that degree–elevated Bernstein representations also have systematically smaller root/value condition numbers than those having the nominal degree of the polynomial under consideration. One is thus lead to enquire as to whether the construction of Bernstein representations of artificially elevated degrees might also constitute a viable "pre–conditioning" strategy. We offer at present just two comments on this: (i) the situation is not entirely analogous with the subdivision case, since degree–elevated representations incur increased *computation* and *data* requirements in their construction; and (ii) for practical use, the subdivision approach is likely to yield greater accuracy improvements, since significant reductions of the root condition numbers through degree elevation require highly–inflated degrees [FR87].

Chapter 5

General Root Finding Concepts

This chapter discusses general concepts used in the polynomial root finding algorithms presented in chapter [§ 6].

Section [§ 5.1] reviews the Bernstein subdivision process along with floating point error accumulation. Section [§ 5.2] describes a modified Horner method for evaluating polynomials in Bernstein form. Section [§ 5.3] outlines the process for deflating a real root from a Bernstein polynomial. Section [§ 5.4] discusses polynomial coefficient normalization for preventing numerical overflow and underflow. Section [§ 5.5] considers approximating zeros at either end control point of the corresponding Bernstein coefficients. Section [§ 5.6] describes differentiation of polynomials in Bernstein form, along with coefficient error accumulation. Section [§ 5.7] reviews schemes for bounding the real roots of a polynomial. Section [§ 5.8] outlines a "pseudo" basis conversion technique for converting between Bernstein and power forms. Section [§ 5.9] discusses closed form schemes for solving the real roots of degree 2 through 4 polynomials.

5.0.1 Pseudo Code

This subsection explains the pseudo code notation and syntax employed in presenting the algorithms in Chapter 6.

The symbol □ denotes a comment, and the BOLDSMALLCAPS font is used for function names and pseudo code control words, e.g., BDEFLATE, FOR, IF, and ENDIF.

Curly brackets { } delimit sets of *expressions* (arrays, equations, etc.) represented by

$$\{ \ldots indexed\ expressions \ldots \}_{index=initial,next}^{limit}$$

where the annotated subscripts are only used in cases to minimize confusion. Lists with multiple expressions are separated by commas. The number of expressions in a list constitutes the size of the list. Lists of similar size may be assigned to each other.

Control loops are designated by either the above annotated lists for simple expressions, or as standard FOR and WHILE statements.

Both $\mathcal{E}_{variable}$ and $\varepsilon_{variable}$ represent error values with respect to the subscripted variable, where $\varepsilon_{variable}$ usually represents coefficient errors such as ε_{y_i}, and $\mathcal{E}_{variable}$ usually represents a global error bound such as $\mathcal{E}_{\|y_i\|}$ or the error of the computed value of an evaluation algorithm such as $\mathcal{E}_{y(t)}$.

5.1 Bernstein Subdivision

Given a Bézier curve $\mathbf{P}_{[a,b]}(t)$ and a real number c, the *de Casteljau algorithm* [BFK84] "subdivides" the curve at $t = c$ by finding the coefficients for $\mathbf{P}_{[a,c]}(t)$ and $\mathbf{P}_{[c,b]}(t)$. In many situations, only one of the two resulting curve segments is needed, and some computation can be saved by only computing the curve segment of interest. Algorithm BSUBDIV$_{left}$ returns $\mathbf{P}_{[a,b]}(t)$ and BSUBDIV$_{right}$ returns $\mathbf{P}_{[c,b]}(t)$. These algorithms also perform linearized running error analysis [§ 3.2.2] for tracking coefficient error that accumulates during the subdivision computation.

The de Casteljau Algorithm

Consider a degree n Bernstein form polynomial represented as an explicit Bézier curve (equation 2.1.2.5)

$$\mathbf{P}_{[a,b]}(t) = (t, y_{[a,b]}(t)) = \sum_{i=0}^{n} \mathbf{P}_i \binom{n}{i} (1-u)^{n-k} u^k \tag{5.1.0.1}$$

where, again, the control points $\mathbf{P}_k$ are evenly spaced along the horizontal axis. $\mathbf{P}_{[a,b]}(t)$ can be subdivided at $t = c$ into $\mathbf{P}_{[a,c]}(t)$ and $\mathbf{P}_{[c,b]}(t)$ as follows. Set $\tau = \frac{c-a}{b-a}$ and add a superscript of 0 to the control points of $\mathbf{P}_{[a,b]}(t)$,

$$\mathbf{P}_i^0 \equiv \mathbf{P}_i, \quad i = 0, \ldots, n.$$

Then compute the auxiliary points

$$\left\{ \left\{ \mathbf{P}_i^j \;=\; (1-\tau)\mathbf{P}_{i-1}^{j-1} + \tau \mathbf{P}_i^{j-1} \right\}_{i=j,j+1}^{n} \right\}_{j=1,2}^{n}. \tag{5.1.0.2}$$

The control points of $\mathbf{P}_{[a,c]}(t)$ are

$$\mathbf{P}_{[a,c]_i} = \mathbf{P}_i^i \tag{5.1.0.3}$$

and the control points of $\mathbf{P}_{[c,b]}(t)$ are

$$\mathbf{P}_{[c,b]_i} = \mathbf{P}_n^{n-i} \tag{5.1.0.4}$$

The de Casteljau algorithm also evaluates ths curve at $\mathbf{P}(c)$ since $\Big(c, y(c)\Big) = \mathbf{P}_n^n$. However, subdivision is an $\mathcal{O}(n^2)$ algorithm, incurring $\frac{1}{2}n(n+1)$ total arithmetic operations, so if evaluation without subdivision is called for, the algorithm in §5.2 is faster.

Figure 5.1 illustrates the de Casteljau algorithm applied to a cubic explicit Bézier curve subdivided at $t = 0.4$.

Note that when the de Casteljau algorithm is applied to an explicit Bézier curve, the control points remain evenly spaced along the t axis: Initially, the control points $\mathbf{P}_{[a,b]_i}$ have t coordinates $a + \frac{i}{n}(b-a)$, and after subdivision, the t coordinates of $\mathbf{P}_{[a,c]_i}$ are $a + \frac{i}{n}(c-a)$ and the t coordinates of $\mathbf{P}_{[c,b]_i}$ are $c + \frac{i}{n}(b-c)$. Thus, the algorithm really only needs to be applied to the y coordinates of the control points.

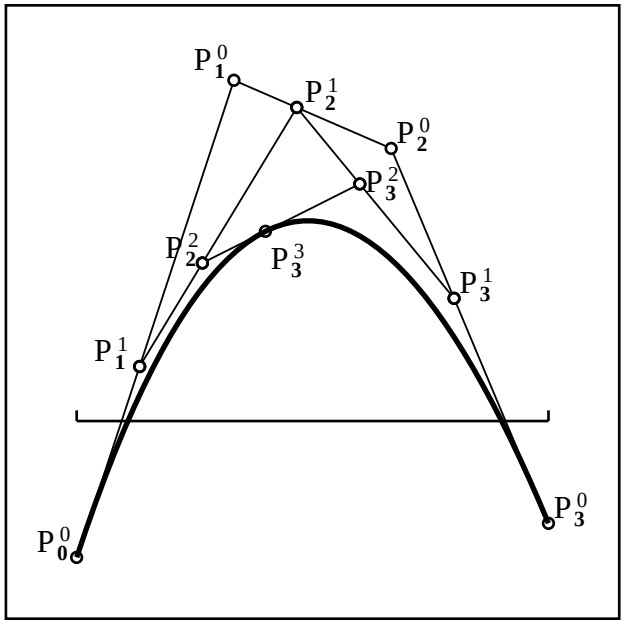

Figure 5.1: Subdividing a cubic explicit Bézier curve.

Subdivision Coefficient Errors

Applying the de Casteljau algorithm to the Bernstein coefficients y_i with $\sigma = (1 - \tau)$ yields the respective intermediate coefficients

$$y_i^j \;=\; \sigma\; y_{i-1}^{j-1} \;+\; \tau\; y_i^{j-1}. \tag{5.1.0.5}$$

Their respective coefficient errors e_i^j [FR87] which represent the accumulated error bounding the computed value y_i^j are derived by applying linearized running error analysis [§ 3.2.2] to (5.1.0.5):

$$e_i^j \;=\; \left(\left|y_i^j\right| + |\sigma| \left|y_{i-1}^{j-1}\right| + |\tau| \left|y_i^{j-1}\right| \right) \eta \;+\; \left(|\sigma|\, e_{i-1}^{j-1} + |\tau|\, e_i^{j-1} \right) \tag{5.1.0.6}$$

with $\{\; e_i^0 = \varepsilon_{y_i} \;\}_{i=0,1}^{n}$ and $\mathcal{E}_{y(t)} \approx e_n^n$, where ε_{y_i} denotes the initial coefficient error of y_i. e_n^n can be used to check for polynomial roots, since $|y(t)| \leq e_n^n$ is as close to zero as we can expect.

The e_i^j for the left and right subintervals (5.1.0.4) are $\varepsilon_{y_{[a,c]_i}} = e_i^i$ and $\varepsilon_{y_{[c,b]_i}} = e_n^{n-i}$,

respectively. If $\sigma = \tau = 1/2$, then expression (5.1.0.5) simplifies, reducing expression (5.1.0.6) to

$$e_i^j = \left(2 \left| y_i^j \right| \right) \eta + \left(e_{i-1}^{j-1} + e_i^{j-1} \right) |t| . \tag{5.1.0.7}$$

Subdivision Global Error Bound

A global bound [FR87] on the magnitude of the round-off error for any computed value $t \in [a, b]$ of equation (5.1.0.5) is given by

$$\mathcal{E}_{y(t)} = 2\ n\ \|y_i\|\ \eta. \tag{5.1.0.8}$$

Here $\|y_i\|$ indicates the greatest absolute value of the coefficient y_i, and η is the machine unit for round-off [§ 3.2.2].

5.1.1 Algorithms: BSUBDIV$_{left}$ and BSUBDIV$_{right}$

Both algorithms presented below use a polynomial structure defined by

$$p = \{[a, b], \{y_i, \varepsilon_{y_i}\}_{i=0}^{n}\}$$

in their argument lists, where $p_{[a,c]}$ and $p_{[c,b]}$ denote the left and right subdivided polynomial segments. BSUBDIV$_{left}$ and BSUBDIV$_{right}$ both return $p_{[a,c]} \equiv p^L$ and $p_{[c,b]} \equiv p^R$. They differ only in that in BSUBDIV$_{left}$, $p_{[a,b]}$ is overwritten by $p_{[c,b]}$ and in BSUBDIV$_{right}$, $p_{[a,b]}$ is overwritten by $p_{[a,c]}$.

Algorithm 5.1.1.1 (BSUBDIV$_{left}$ $(c, p^{(L)}, p^R)$) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *subdivided at* $c \in [a, b]$ *are replaced with the coefficients defined over the left subinterval* $[a, c]$. *The coefficients for the polynomial defined over the right subinterval* $[c, b]$ *are appropriately assigned and returned.*

1. $t = (c - a)/(b - a)$
2. $s = 1 - t$
3. $\{\ y_{[c,b]_n}, \varepsilon_{y_{[c,b]_n}}\ \} = \{\ y_n, \varepsilon_{y_n}\ \}$
4. FOR $(j = 1, 2, \ldots, n)$

5. FOR $(i = n, n-1, \ldots, j)$
6. $\quad y_{tmp} \;=\; y_i$
7. $\quad y_i \;=\; s\; y_{i-1} \;+\; t\; y_{tmp}$
8. $\quad \varepsilon_{y_i} \;=\; (\;|y_i| \;+\; |s|\;|y_{i-1}| \;+\; |t|\;|y_{tmp}|\;)\;\eta \;+\; (\;|s|\;\varepsilon_{y_{i-1}} \;+\; |t|\;\varepsilon_{y_i}\;)$

ENDFOR$_i$

9. $\{\; y^R_{n-j}, \varepsilon_{y^R_{n-j}} \;\} \;=\; \{\; y_n, \varepsilon_{y_n} \;\}$

ENDFOR$_j$

END OF BSUBDIV$_{left}$

Algorithm 5.1.1.2 (BSUBDIV$_{right}$ $(t, p^{(R)}, p^L)$) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *subdivided at* $t \in [a,b]$ *are replaced with the coefficients defined over the right subinterval* $[c,b]$. *The coefficients* $\{y^L_i, \varepsilon_{y^L_i}\}$ *defined over the left subinterval* $[a,c]$ *are appropriately assigned and returned.*

1. $t = (c-a)/(b-a)$
2. $s = 1 - t$
3. $\{\; y^L_0, \varepsilon_{y^L_0} \;\} \;=\; \{\; y_0, \varepsilon_{y_0} \;\}$
4. FOR $(j = 1, 2, \ldots, n)$
5. $\quad$ FOR $(i = 0, 1, \ldots, j)$
6. $\quad\quad y_{tmp} \;=\; y_i$
7. $\quad\quad y_i \;=\; s\; y_{tmp} \;+\; t\; y_{i+1}$
8. $\quad\quad \varepsilon_{y_i} \;=\; (\;|y_i| \;+\; |s|\;|y_{tmp}| \;+\; |t|\;|y_{i+1}|\;)\;\eta \;+\; (\;|s|\;\varepsilon_{y_i} \;+\; |t|\;\varepsilon_{y_{i+1}}\;)$

$\quad$ ENDFOR$_i$

9. $\quad \{\; y^L_j, \varepsilon_{y^L_j} \;\} \;=\; \{\; y_0, \varepsilon_{y_0} \;\}$

ENDFOR$_j$

END OF BSUBDIV$_{right}$

5.1.2 Numerical Stability

The de Casteljau algorithm using finite precision arithmetic is numerically stable for values of $t \in [a,b]$, but can be unstable otherwise. This can be seen from equation (5.1.0.6). For $t \notin [a,b]$, $|\sigma| > 1$ or $|\tau| > 1$ (or both), and e^j_i can become much larger than e^{j-1}_{i-1} or e^{j-1}_i, while for $t \in [a,b]$, e^j_i is roughly the same magnitude as e^{j-1}_{i-1} or e^{j-1}_i.

5.2 Bernstein Modified Horner Evaluation

This section reviews the evaluation of a polynomial expressed in Bernstein form using a *modified Horner's algorithm* [Pav82]. This algorithm is called BEVAL and includes linearized running error analysis.

Horner's algorithm adapted to the Bernstein representation can evaluate a polynomial with roughly the same numerical precision as the subdivision algorithm [§ 5.1], yet with fewer operations. However, neither algorithm can claim superior error bounds in the evaluation of arbitrary polynomials [FR88].

5.2.1 Horner's Method Expressed in Power Form

Horner's method for a polynomial in power form $p(t) = \sum_{i=0}^{n} p_i t^i$ evaluates a polynomial using *nested multiplication*, which is the most efficient method of evaluating a power form polynomial at a single value [MB72]. It requires $\mathcal{O}(n)$ steps (n multiplies and n adds).

$$\{\ \hat{p}_i \ = \ t\ \hat{p}_{i-1} \ + \ p_{n-i}\ \}_{i=1,2}^{n}, \qquad \hat{p}_0 = p_n, \qquad p(t) \equiv \hat{p}_n \tag{5.2.1.9}$$

$\{\hat{p}_i\}_{i=0,1}^{n-1}$ are the deflated polynomial coefficients when t is a root of $p(t)$. Applying linearized running error analysis [§ 3.2.2] to equation (5.2.1.9), which has coefficients p_i with corresponding initial errors ε_{p_i}, yields the corresponding intermediate error values denoted by $\hat{e}_i$. We neglect any error in the floating point representation of t.

$$\{\ \hat{e}_i = (|\hat{p}_i| + |t|\ |\hat{p}_{i-1}|)\eta + (|t|\ \hat{e}_{i-1} + \varepsilon_{p_{n-i}})\ \}_{i=1,2}^{n}, \qquad \hat{e}_0 = \varepsilon_{p_n}, \qquad \hat{e}_n = \mathcal{E}_{p(t)} \tag{5.2.1.10}$$

where η is the machine round-off unit [§ 3.2.2]. $\hat{e}_i$ represents the maximum error bounding $\hat{p}_i$. The first term tracks the expression's round-off error and the second term tracks the inherent error due to both the expression as well as the coefficient errors ε_{p_i}.

Expanding the upper error bound for the Horner evaluation given by Farouki and Rajan

[FR87] to include the inherent coefficient errors yields the following global bound

$$\mathcal{E}_{\|\hat{p}_i\|} \;\leq\; 2\; n\; \|\hat{p}_i\|\; \eta \;+\; n\; \|\varepsilon_{p_i}\| \tag{5.2.1.11}$$

where $\|\hat{p}_i\|$ and $\|\varepsilon_{p_i}\|$ represent the absolute magnitude of the intermediate coefficients and of the original coefficient errors, respectively.

5.2.2 Horner's Method Modified for Bernstein Form

Horner's scheme can be adapted to evaluate a degree n polynomial in Bernstein form [§ 2.1.2]

$$y_{[a,b]}(t) \;=\; \sum_{i\,=\,0}^{n} y_i \binom{n}{i} (1-t)^{n-i} t^i$$

by dividing by $(1-t)^n$ [Pav82]. This yields the expression

$$\frac{\mathrm{y(t)}}{(1-\mathrm{t})^{\mathrm{n}}} = \left(\mathrm{y}_0 \;+\; \frac{\mathrm{n}}{1}\mathrm{v}\left(\mathrm{y}_1 \;+\; \frac{\mathrm{n}-1}{2}\mathrm{v}\left(\mathrm{y}_2 \;+\; \frac{\mathrm{n}-2}{3}\mathrm{v}(\mathrm{y}_3 \;+\ldots+\; \frac{2}{\mathrm{n}-1}\mathrm{v}(\mathrm{y}_{\mathrm{n}-1} \;+\; \frac{1}{\mathrm{n}}\mathrm{vy}_{\mathrm{n}})\ldots)\right)\right)\right) \tag{5.2.2.12}$$

with $v = \frac{t}{1-t}$ for $t \neq 1$ which is appropriate over the interval $[0, \frac{1}{2}]$. Factoring out the quantity t^n yields the expression

$$\frac{\mathrm{y(t)}}{\mathrm{t}^{\mathrm{n}}} = \left(\mathrm{y}_{\mathrm{n}} \;+\; \frac{\mathrm{n}}{1}\mathrm{u}\left(\mathrm{y}_{\mathrm{n}-1} \;+\; \frac{\mathrm{n}-1}{2}\mathrm{u}\left(\mathrm{y}_{\mathrm{n}-2} \;+\; \frac{\mathrm{n}-2}{3}\mathrm{u}(\mathrm{y}_{\mathrm{n}-3} \;+\ldots+\; \frac{2}{\mathrm{n}-1}\mathrm{u}(\mathrm{y}_1 \;+\; \frac{1}{\mathrm{n}}\mathrm{uy}_0)\ldots)\right)\right)\right) \tag{5.2.2.13}$$

with $u = \frac{1-t}{t}$ for $t \neq 0$ which is appropriate over the interval $[\frac{1}{2}, 1]$.

Equation (5.2.2.13) can be re-written

$$\left\{\; \hat{y}_i \;=\; y_i \;+\; \hat{y}_{i-1}\; u\; \frac{i}{n-i+1} \;\right\}_{i=1,2}^{n}, \qquad \hat{y}_0 = y_0, \qquad \hat{y}_n\; (1-t)^n \equiv y(t). \tag{5.2.2.14}$$

This requires n steps with a total $6n$ operation per evaluation. However, we typically apply the modified Horner algorithm several times to a given polynomial, and it pays to simply pre-multiply each y_i by $\binom{n}{i}$. Once this is done, $y(t)$ can be evaluated in $3n+1$ operations. Applying linearized running error analysis to equation (5.2.2.14) (neglecting any inherent error in the value t) it is possible to factor the term corresponding to the round-off bound

as

$$\left\{ \hat{er}_i \;=\; |y_i| \;+\; (\hat{er}_{i-1} \;+\; 5\,|\hat{y}_{i-1}|)\; u\; \frac{i}{n-i+1} \right\}_{i=1,2}^{n}, \qquad \hat{er}_0 = 0. \tag{5.2.2.15}$$

The same analysis gives inherent error bound term as

$$\left\{ \hat{ei}_i \;=\; \varepsilon_{y_i} \;+\; \hat{ei}_{i-1}\; u\; \frac{i}{n-i+1} \right\}_{i=1,2}^{n}, \qquad \hat{ei}_0 = \varepsilon_0. \tag{5.2.2.16}$$

This yields the total error bounding the computed value $y(t)$ as

$$\mathcal{E}_{y_{[a,b]}(t)} \;\leq\; (\;(\;\hat{er}_n \;+\; n\,|\hat{y}_n|\;)\;\eta \;+\; \hat{ei}_n\;)\;t^n \tag{5.2.2.17}$$

which includes error analysis due to the operations involving the quantity t^n.

Thus, the evaluation of a Bernstein form polynomial using the above modified Horner method maintains $\mathcal{O}(n)$ complexity with a total of $(7n+1)$ arithmetic operations for computing $y(t) = \hat{y}_n\; t^n$. Additional $(6n+5)$ operations are required for computing $\mathcal{E}_{y(t)}$, where $4n$ and $2n$ arithmetic operations are attributed to the round-off and inherent error bound terms, respectively.

5.2.3 Algorithm: BEVAL

The algorithm outlined below uses a variable solution list argument defined by

$$s \;=\; \left\{\; [a,b],\; \{y_i, e_{y_i}\}_{i=0}^{n},\; y(t),\; \mathcal{E}_{y(t)},\; y'(t) \;\right\}$$

where $y(t)$ is the computed value, $\mathcal{E}_{y(t)}$ is the computed error bounding this value, and $y'(t)$ is the computed derivative.

Algorithm 5.2.3.1 (BEVAL (t, s)) *The initial coefficients $\{y_i, \varepsilon_{y_i}\}$ are evaluated at $t \in [a,b]$; the computed value $y(t)$, the computed error $\mathcal{E}_{y(t)}$ bounding $y(t)$, and the computed derivative $y'(t)$ are returned.*

1. $scale \;=\; 1$
2. $e_r \;=\; 0$
3. $\tau \;=\; (t-a)\;/\;(b-a)$
4. IF $(\tau < \frac{1}{2})$

5. $v = \frac{\tau}{1-\tau}$
6. $e_i = \varepsilon_n$
7. $a = y_n$
8. $b' = y_n - y_{n-1}$
9. FOR $(i = n-1, n-2, \ldots, 0)$
10. $scale = scale\ (1-\tau)$
11. $b = y_i + a\ v\ \frac{n-i}{i+1}$
12. $e_r = |b| + (e_r + 5\ |a|)\ v\ \frac{n-i}{i+1}$
13. $e_i = \varepsilon_i + e_i\ v\ \frac{n-i}{i+1}$
14. IF $(i < n-1)$ $b' = (y_{i+1} - y_i) + b'\ v\ \frac{n-1-i}{i+1}$
15. $a = b$

ENDFOR$_i$

16. $n_{scale} = 2n-1$
17. $scale' = \frac{scale}{1-\tau}$
18. ELSE
19. $u = \frac{1-\tau}{\tau}$
20. $e_i = \varepsilon_0$
21. $a = y_0$
22. $b' = y_1 - y_0$
23. FOR $(i = 1, 2, \ldots, n)$
24. $scale = scale\ \tau$
25. $b = y_i + a\ u\ \frac{i}{n-i+1}$
26. $e_r = |b| + (e_r + 5\ |a|)\ u\ \frac{i}{n-i+1}$
27. $e_i = \varepsilon_i + e_i\ u\ \frac{i}{n-i+1}$
28. IF $(i < n)$ $b' = (y_{i+1} - y_i) + b'\ u\ \frac{i}{n-i}$
29. $a = b$

ENDFOR$_i$

30. $n_{scale} = n$
31. $scale' = \frac{scale}{\tau}$

ENDIF

32. $y(t) = b\ scale$
33. $\mathcal{E}_{y(t)} =$ ERR $(\ldots)$
34. $y'(t) = n\ b'\ scale'$

END OF BEVAL

The variables e_r and e_i denote round-off and inherent error bound terms, respectively. The function ERR $(\ldots)$ represents the error bounding the computed value. It is denoted

by

$$\text{ERR}\ (\ldots)\ =\ ((e_r\ +\ n_{scale}\ |b|)\ \eta\ +\ e_i)\ scale$$

when including the coefficient errors ε_{y_i}, and by

$$\text{ERR}\ (\ldots)\ =\ (e_r\ +\ n_{scale}\ |b|)\ \eta\ scale$$

when neglecting the coefficient errors ε_{y_i}.

5.2.4 Polynomial Evaluation Considerations

Note that for root finding applications it is possible to disregard the *scale* factor variables since $y(t) = 0$ yields $\frac{y(t)}{scale} = 0$. Thus, any expression involving related *scale* factor variables might be eliminated, and the corresponding ERR function adjusted accordingly. However, initial testing of this idea suggested that the simplification can impede convergence, and we have opted against using it.

Although implementation of running error analysis provides tighter error bounds for computed values, the number of arithmetic operations and complexity of the algorithm doubles.

Scaled Bernstein Polynomial Coefficients

An important alteration which helps increase efficiency and minimize roundoff is to eliminate the fractional factor from STEPS 11-14 (and corresponding STEPS 25-28) by first scaling each of the Bernstein coefficients by their appropriate binomial factor represented by

$$\hat{y}_i = y_i \binom{n}{i} \tag{5.2.4.18}$$

which creates *scaled polynomial coefficients* $\hat{y}_i$ [FR88] which are explicitly used for the evaluation of Bernstein form polynomials. The corresponding round-off expression in in STEPS 12 and 26 reduces to

$$\hat{e}_r\ =\ |b|\ +\ |\hat{y}_i|\ +\ (\hat{e}_r\ +\ 3\ |a|)\ t \tag{5.2.4.19}$$

and the inherent error expression in STEPS 13 and 27 reduces to

$$\hat{e}_i \;=\; \hat{\varepsilon}_i \;+\; \hat{e}_i \; t. \tag{5.2.4.20}$$

It is important to not that this alteration cannot be employed for isolation concepts and their respective functions dependent on the convex hull property, because this property does not hold for the scaled polynomial coefficients.

5.3 Bernstein Deflation

This section describes the process of deflating a zero from a Bernstein form polynomial. Three *Bernstein deflation algorithms*, BDEFLATE$_t$, BDEFLATE$_{left}$, and BDEFLATE$_{right}$, are presented that deflate roots located at any parameter value as well as the special cases when roots exist at either end control point. The algorithms track coefficient error accumulation by applying linearized running error analysis to the deflation computation.

5.3.1 Preliminary Concepts

Deflation [PW71, Ric83, Sed93] is the process of computing

$$y_{n-1}(t) = \frac{y_n(t)}{(\tau - 1)t \;+\; \tau(1-t)}$$

where $y_n(t)$ is a degree n Bernstein form polynomial [§ 2.1.2] with a root at $t = \tau$. $y_{n-1}(t)$ contains all the roots of $y_n(t)$ except τ.

5.3.2 Algorithms: BDEFLATE$_t$, BDEFLATE$_{left}$, and BDEFLATE$_{right}$

All three algorithms presented below use the structure defined by

$$p \;=\; \{[a,b], \{y_i, \varepsilon_{y_i}\}_{i=0}^{n}\}$$

in their argument lists.

Algorithm 5.3.2.1 (BDEFLATE$_t$ (t,p)) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *deflated at* $t \in [a,b]$ *are replaced with their deflated values.*

1. $\tau \;=\; (t-a) \;/\; (b-a)$
2. IF $(\tau < \frac{1}{2})$
3. $\quad u \;=\; \tau \;/\; (1-\tau)$
4. $\quad y_{tmp1} \;=\; y_{n-1}$
5. $\quad \varepsilon_{y_{tmp1}} \;=\; \varepsilon_{y_{n-1}}$
6. $\quad y_{n-1} \;=\; y_n$
7. $\quad$ FOR $(i = 1, 2, \ldots, n-1)$
8. $\quad\quad y_{tmp2} \;=\; y_{n-i-1}$
9. $\quad\quad \varepsilon_{y_{tmp2}} \;=\; \varepsilon_{y_{n-i-1}}$
10. $\quad\quad y_{n-i-1} \;=\; (n\; y_{tmp1} \;+\; i\; y_{n-i}\; u) \;/\; (n-i)$
11. $\quad\quad \varepsilon_{y_{n-i-1}} \;=\; \text{ERR}(y_{n-i}, y_{tmp1}, \varepsilon_{y_{n-i}}, \varepsilon_{y_{tmp1}})$
12. $\quad\quad y_{tmp2} \;=\; y_{tmp1}$
13. $\quad\quad \varepsilon_{y_{tmp2}} \;=\; \varepsilon_{y_{tmp1}}$

    $\quad$ ENDFOR$_i$
14. ELSE
15. $\quad u \;=\; (1-\tau) \;/\; \tau$
16. $\quad$ FOR $(i = 1, 2, \ldots, n-1)$
17. $\quad\quad y_{tmp} \;=\; y_i$
18. $\quad\quad \varepsilon_{y_{tmp}} \;=\; \varepsilon_{y_i}$
19. $\quad\quad y_i \;=\; (n\; y_{tmp} \;+\; i\; y_{i-1}\; u) \;/\; (n-i)$
20. $\quad\quad \varepsilon_{y_i} \;=\; \text{ERR}(y_i, y_{tmp}, \varepsilon_{y_i}, \varepsilon_{y_{tmp}})$

    $\quad$ ENDFOR$_i$

    ENDIF
21. $n \;=\; n-1$

END OF BDEFLATE$_t$

BDEFLATE$_t$ approximates the cumulative coefficient error by applying linearized running error analysis [§ 3.2.2] to STEPS 10 and 19, yielding the following error

$$\text{ERR}(y_j, y_{tmp}, \varepsilon_{y_j}, \varepsilon_{y_{tmp}}) \;=\; \left(2|y_j| \;+\; \frac{n|y_{tmp}| \;+\; 3j|y_{j-1}|\;|u|}{n-j}\right)\eta \;+\; \frac{n\;\varepsilon_{y_{tmp}} \;+\; j\;\varepsilon_{y_{j-1}}}{n-j}$$

where the 1st and 2nd terms account for the round-off and inherent errors, respectively.

Algorithm 5.3.2.2 (BDEFLATE$_{left}$ (p)) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *deflated at* $t = a$ *are replaced with their deflated values.*

1. FOR $(i = 0, 1, \ldots, n-1)$
2. $\quad y_i \;=\; y_{i+1}\ \frac{n}{i+1}$
3. $\quad \varepsilon_{y_i} \;=\; 2\ |y_i|\ \eta \;+\; \varepsilon_{y_{i+1}}\ \frac{n}{i+1}$

   ENDFOR$_i$
4. $n \;=\; n-1$

END OF BDEFLATE$_{left}$

Algorithm 5.3.2.3 (BDEFLATE$_{right}$ (p)) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *deflated at* $t = b$ *are replaced with their deflated values.*

1. FOR $(i = 0, 1, \ldots, n-1)$
2. $\quad y_i \;=\; y_i\ \frac{n}{n-i}$
3. $\quad \varepsilon_{y_i} \;=\; 2\ |y_i|\ \eta \;+\; \varepsilon_{y_i}\ \frac{n}{n-i}$

   ENDFOR$_i$
4. $n \;=\; n-1$

END OF BDEFLATE$_{right}$

5.3.3 Deflation Considerations

Deflation reduces the polynomial degree, as well as preventing double convergence to a multiple root when used prudently [PFTV90].

Deflation is *stable* when the roots are deflated in order of ascending magnitude [PFTV90, Pet89, Ric83, Hil74, Wil63]. Deflation is *unstable* when a root is approximated inaccurately, and thus, induces errors in the deflated coefficients. Repeatedly performed, this yields more inaccurate roots and perturbed coefficients which may be compounded to yield meaningless results. This is especially evident in successive approximation root finding algorithms [§ 2.3.2].

It is recommended [PFTV90, Ric83, Hil74, RR78, Hen64] that, whenever possible, a root from a deflated polynomial should be *purified*. Purification means using the root as an initial guess to a refining process such as Newton's method for a couple of iterations on the original (undeflated) polynomial, before deflation of the root.

A general guideline which has been validated empirically is to *avoid* deflation whenever

possible, because when an imprecise root is deflated, the accuracy of subsequent approximated roots will suffer.

5.4 Polynomial Coefficient Normalization

This section presents two algorithms that perform coefficient normalization to guard against floating point overflow and underflow. NORMPOLY$_{base}$ combines concepts from [Jen75, Mad73, Dun72] and normalizes the polynomial coefficients without inducing roundoff by computing a scale factor which is the reciprocal of the largest power of the machine base, or radix, not exceeding the largest modulus of the coefficients. NORMPOLY$_{max}$ normalizes the polynomial with a scale factor that is merely the reciprocal of the largest modulus of the coefficients.

5.4.1 Preliminary Concepts

Coefficient normalization [FR88, FL85, Jen75, Mad73, Dun72, Ham71] is the process by which a suitable scale factor is applied to the polynomial coefficients in order to avoid floating-point overflow and underflow.

Various coefficient scaling methods have been implemented to minimize round-off error in the computed roots. Schemes that optimize the variation of magnitude in the coefficients are described by [GK90, Dun74]. Results from applying this technique to a general root finding method (such as Laguerre's method) indicates no significant effect on the precision of the approximate roots [GK90]. Although Dunaway [Dun74] reports that this technique does improve GCD sequence determination — a claim that invites further scrutiny.

Strategies that scale the coefficients by a power of the machine radix are implemented in Jenkin's RPOLY [Jen75] and [Mad73, Dun72], and do not induce round-off error. A polynomial coefficient y_i is represented in normalized floating-point arithmetic as

$$y \;=\; m\;\beta^{e}$$

where e is the exponent and m is the mantissa. It can be scaled by a factor $s = \beta^{\pm p}$, yielding

$$y\ s\ =\ m\ \beta^{e}\ \beta^{\pm p}\ =\ m\ \beta^{(e\ \pm\ p)}$$

which preserves all base digits of the mantissa and updates only the exponent by power p, and thus prevents round-off.

Another method outlined in [FR88] scales Bernstein coefficients, yielding a polynomial with a unit root–mean–square value on $[0, 1]$.

The only scaling that Farmer-Loizou [FL85] deemed necessary was the initial scaling of the polynomial to monic form, followed by deflation of any zeros at the origin.

5.4.2 Algorithms: NORMPOLY$_{base}$ and NORMPOLY$_{max}$

Algorithm 5.4.2.1 (NORMPOLY$_{base}$ (y_{tol}, $\{y_i, \varepsilon_{y_i}\}_{i=0}^{n}$)) *The coefficients y_i, ε_{y_i} and their global error bound y_{tol} are normalized and returned.*

1. $\{s\ =\ \|y_i\|\ \}_{i=0}^{n}$
2. $p\ =\ \text{LOG}_{10}(s)\ /\ \text{LOG}_{10}(b)$
3. $s\ =\ b^{-p}$
4. $\{\ y_i\ =\ y_i\ s,\ \ \varepsilon_{y_i}\ =\ \varepsilon_{y_i}\ s\ \}_{i=0,1}^{n}$
5. $y_{tol}\ =\ s\ y_{tol}$

END OF NORMPOLY$_{base}$

Algorithm 5.4.2.2 (NORMPOLY$_{max}$ (y_{tol}, $\{y_i, \varepsilon_{y_i}\}_{i=0}^{n}$)) *The coefficients y_i, ε_{y_i} and their global error bound y_{tol} are normalized and returned.*

1. $\{\ y_i\ =\ \frac{y_i}{\|y_i\|},\ \ \varepsilon_{y_i}\ =\ \frac{\varepsilon_{y_i}}{\|y_i\|}\ \}_{i=0,1}^{n}$
2. $y_{tol}\ =\ \frac{y_{tol}}{\|y_i\|}$

END OF NORMPOLY$_{max}$

5.4.3 Normalization considerations

Coefficient normalization is implemented in the test driver module that generates the coefficients from the specified roots, where the product terms are normalized at each iteration.

Without such normalization, overflow was experienced when trying to generate high degree polynomials. The root finding algorithms only normalize the initial input coefficients, and after deflating zeros at endpoints [§5.5].

Both algorithms work equally well.

5.5 Bernstein End Control Point Root Approximation

This section presents two algorithms (BEND0$_{left}$ and BEND0$_{right}$) which detect roots of Bernstein form polynomials at the left and right limits of their parameter domains.

The end control points of an explicit Bézier curve [§ 2.1.2] interpolate the curve:

$$\mathbf{P}_{[a,b]_0} \;=\; \mathbf{P}_{[a,b]}(a) \;=\; (a, y_0), \qquad \underline{\mathrm{P}}_{[a,b]_n} = \mathbf{P}_{[a,b]}(b) \;=\; (b, y_n).$$

When $|y_0| \leq \varepsilon$ or $|y_n| \leq \varepsilon$, a or b are taken to be roots of $\mathbf{P}(t)$. The first step of most of our algorithms is to check for roots at a or b, deflate them if they exist [§ 5.3], find a new coefficient error bound y_{tol} [§ 5.1], and re–normalize [§ 5.4]. Figure 5.2 shows the explicit Bézier curves before and after both roots are deflated at $t = a$ and $t = b$.

5.5.1 Algorithms: BEND0$_{left}$ and BEND0$_{right}$

Algorithm 5.5.1.1 (BEND0$_{left}$ ($\{\ y_{tol},\ \{y_i, \varepsilon_{y_i}\}_{i=0,1}^{n}\},\ n_{roots}$)) *Bernstein coefficients of the polynomial $y(t)_{[a,b]}$ are tested until no root(s) exist at $t = a$. For each root detected at a, the root is deflated, a new global coefficient error bound y_{tol} is computed, and then normalized along with the coefficients $\{y_i\}$ and their errors $\{\varepsilon_{y_i}\}$.*

1. $n_{roots} \;=\; 0$
2. $\tau \;=\; 0.$
3. WHILE ($y_{[0,1]}(\tau)$ <= ε AND $n \neq 0$)
4. $\quad n_{roots} = n_{roots} + 1$
5. $\quad Bdeflate_{left}\ (\tau,\ \{y_i, \varepsilon_{y_i}\})$
6. $\quad y_{tol} \;=\; \left\| \ \|\varepsilon_{y_i}\ \|,\ \varepsilon_{\|y_i\|} \ \right\|$
7. $\quad NormPoly\ (y_{tol},\ \{y_i, \varepsilon_i\})$

ENDWHILE

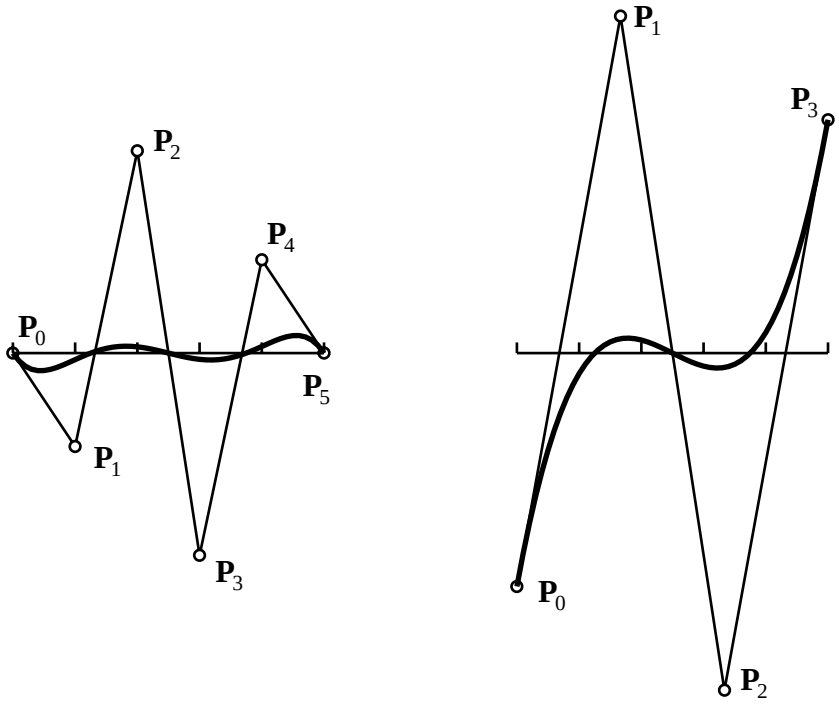

Figure 5.2: Explicit Bézier curves before and after deflating roots at $t = a, b$

END OF BEND0$_{left}$

The tolerance ε in step 3 is the computed error $\mathcal{E}_{y(t)}$ from the Bernstein modified Horner evaluation algorithm [§ 5.2]

$$\text{BEVAL } (t,\ \left\{\ \{y_i,\ \varepsilon_{y_i}\},\ y(t),\ \mathcal{E}_{y(t)}\ \right\})$$

— see considerations below.

Algorithm 5.5.1.2 (BEND0$_{right}$ ($\{\ y_{tol},\ \{y_i, \varepsilon_{y_i}\}_{i=0,1}^{n}\},\ n_{roots}$)) *Bernstein coefficients of the polynomial $y(t)_{[a,b]}$ are tested until no root(s) exist at $t = b$. For each root detected at b, the root is deflated, a new global coefficient error bound y_{tol} is computed, and then normalized along with the coefficients $\{y_i\}$ and their errors $\{\varepsilon_{y_i}\}$.*

□ The algorithm is the same as BEND0$_{left}$ except 2. and 5. are replaced by:

2. $\tau\ =\ 1.$
5. $Bdeflate_{right}\ (\tau,\ \{y_i, \varepsilon_{y_i}\})$

5.5.2 Bernstein End Zero Considerations

Step 3 utilizes $\varepsilon = \mathcal{E}_{y(\tau)}$ which yields a tighter root bound than $\varepsilon = \varepsilon_{\|y_i\|}$. This is illustrated by the case shown in Figure 5.3 with given roots at r_i = { -2.0, 0.9999999999999, 1.0, 1.0, 1.0000000000001, 2.0 }$_1^6$ where there exist clustered roots at the right limit, $b = 1.0$

Using the global bound $\varepsilon = \varepsilon_{\|y_i\|}$ in step 3 yields the 4 roots r_i = { 0.9999999999999989, 1.0, 1.0, 1.0 }$_1^4$; using the computed error bounding the computed value of the modified Bernstein Horner algorithm $\varepsilon = \mathcal{E}_{y(\tau)}$ yields the 3 approximate roots r_i = { 0.9999999999999989, 1.0, 1.0 }$_1^3$.

5.6 Bernstein Differentiation

This section presents two *Bernstein derivative algorithms* (BDERIV and BDERIV$_{pseudo}$) that compute normal and scaled, or pseudo, derivative coefficients. They also account for coefficient error propagation by applying linearized running error analysis to the derivative computation.

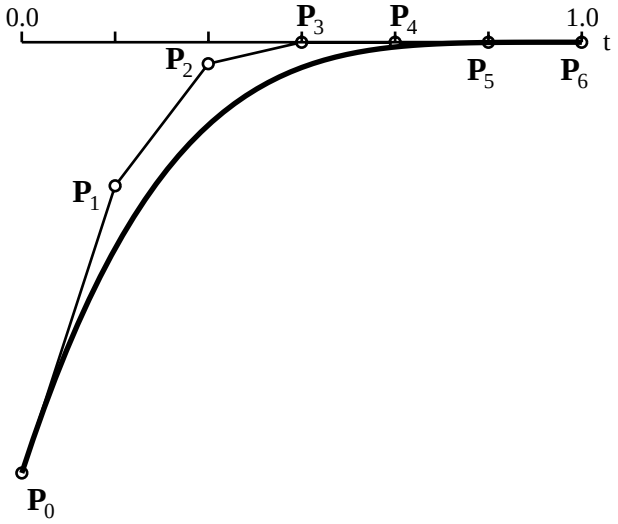

Figure 5.3: Explicit Bézier curve $y_{[0,1]}(t)$ with a cluster of roots at $t = b$

Differentiation of a degree n Bernstein form polynomial [§ 2.1.2] given by

$$y(t) \;=\; \sum_{i=0}^{n} \; y_i \; \binom{n}{i} \; (1-t)^{n-i} \; t^i$$

whose coefficients $\{y_i\}$ have initial errors $\{\varepsilon_{y_i}\}$ yields derivative coefficients represented by

$$\left\{ \; y'_i = (y_{i+1} \;-\; y_i) \; n, \quad \varepsilon_{y'_i} = 2|y'_i|\eta \;+\; n \; (\varepsilon_{y_{i+1}} \;+\; \varepsilon_{y_i}) \; \right\}_{i=0,1}^{n-1} .$$

Each ε'_{y_i} is computed by applying linearized running error analysis to their respective y'_i computation.

Since roots of a polynomial are unaltered when the coefficients of the polynomial are scaled, *pseudo derivative coefficients* are as follows:

$$\left\{ \; y'_{p_i} = (y_{i+1} \;-\; y_i) \; n, \quad \varepsilon_{y'_{p_i}} = |y'_{p_i}|\eta \;+\; (\varepsilon_{y_{i+1}} \;+\; \varepsilon_{y_i}) \; \right\}_{i=0,1}^{n-1} .$$

5.6.1 Algorithms: BDERIV and BDERIV$_{pseudo}$

Both algorithms below use in their argument lists the following two structures

$$p \;=\; \left\{ [a,b], \{y_i, \varepsilon_{y_i}\}_{i=0}^{n} \right\} \quad \text{and} \quad p' \;=\; \left\{ [a',b'], \left\{ y'_i, \varepsilon_{y'_i} \right\}_{i=0}^{n} \right\} .$$

Algorithm 5.6.1.1 (BDERIV (p, p')) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *are differentiated, and their derivative values are returned as* $\{y'_i, \varepsilon_{y'_i}\}$.

1. FOR $(i = 0, 1, \ldots, n-1)$
2. $\quad y'_i \;=\; n\ (y_{i+1} \;+\; y_i)$
3. $\quad \varepsilon_{y'_i} \;=\; 2\ |y'_i|\ \eta \;+\; n\ (\varepsilon_{y_{i+1}} \;+\; \varepsilon_{y_i})$

   ENDFOR$_i$
4. $n \;=\; n-1$
5. $[a', b'] = [a, b]$

END OF BDERIV

Algorithm 5.6.1.2 (BDERIV$_{pseudo}$ (p, p')) *The initial coefficients* $\{y_i, \varepsilon_{y_i}\}$ *are differentiated, and their scaled derivative values returned as* $\{y'_{p_i}, \varepsilon_{y'_{p_i}}\}$.

1. FOR $(i = 0, 1, \ldots, n-1)$
2. $\quad y'_{p_i} \;=\; (y_{i+1} \;+\; y_i)$
3. $\quad \varepsilon_{y'_{p_i}} \;=\; |y'_{p_i}|\ \eta \;+\; (\varepsilon_{y_{i+1}} \;+\; \varepsilon_{y_i})$

   ENDFOR$_i$
4. $n \;=\; n-1$
5. $[a', b'] = [a, b]$

END OF BDERIV$_{pseudo}$

5.6.2 Differentiation Considerations

BDERIV is used in cases where the actual derivative is required such as in Newton root approximation. BDERIV$_{pseudo}$ is utilized during root isolation when the relative magnitude of the coefficients are immaterial.

Error analysis is employed in cases whose computations involve, and consequently results depend, on inherent error accumulation. A significant example is the evaluation of the master polynomial and its derivatives during root approximation, where approximated error of the computed value is required for termination criteria. Note that the approximated error in STEP 3 of BDERIV$_{pseudo}$ provides a tighter coefficient error than the same expression in BDERIV.

5.7 Polynomial Real Root Bounds

Since polynomials in Bernstein form are usually defined over the unit interval [§§ 2.1.2], and one is generally interested in roots on that interval, root bounds are unnecessary for algorithms based on the Bernstein form. The intersection of the convex hull with the t-axis provides tighter root bounds than 0 and 1 if desired. Global root bounds for polynomials in power form have been discussed in [§§ 2.2.1].

5.8 Pseudo Bernstein Basis Conversion

This section presents a stable method of converting between Bernstein and power polynomial forms by defining a pseudo-basis conversion. Two algorithms called ConvCoefs$_{b2\hat{p}}$ and ConvCoefs$_{p2\hat{b}}$ are outlined which convert Bernstein coefficients to pseudo-power coefficients and power coefficients to pseudo-Bernstein coefficients, respectively. The required remapping of the roots generated from the pseudo-coefficients to their proper values is facilitated by the two functions ConvRoots$_{\hat{b}}$ and ConvRoots$_{\hat{p}}$.

5.8.1 Preliminary Concepts

Methods for converting between Bernstein and power forms have been documented in [FR87, LR81, Riv70, CS66] and presented in [§ 3.4]. A stable process of converting coefficients between the Bernstein and power forms became expedient in order to more accurately compare both types of polynomial root finders. A pseudo-basis conversion provides a stable alternative for converting between power and Bernstein coefficients.

Given a Bernstein form polynomial [§ 2.1.2] represented by

$$b_{[0,1]}(t) \;=\; \sum_{i=0}^{n} \; b_i \; \binom{n}{i} \; (1-t)^{n-i} \; t^i, \tag{5.8.1.21}$$

we rewrite it in nested form [§ 5.2] as

$$b_{[0,1]}(t) \;=\; (1-t)^n \;\sum_{i=0}^{n}\; b_i \;\binom{n}{i} \;\left(\frac{t}{1-t}\right)^i. \tag{5.8.1.22}$$

A power form representation is denoted by

$$p(x) \;=\; \sum_{i=0}^{n}\; p_i \;x^i. \tag{5.8.1.23}$$

Excluding the quantity $(1-t)^n$ a pseudo-basis conversion from Bernstein-to-power form is evidently given by

$$p_i \;=\; b_i \;\binom{n}{i} \quad \textit{and} \quad x^i \;=\; \left(\frac{t}{1-t}\right)^i \tag{5.8.1.24}$$

and from power-to-Bernstein form by

$$b_i \;=\; p_i \;/\; \binom{n}{i} \quad \textit{and} \quad t^i \;=\; \left(\frac{x}{1+x}\right)^i. \tag{5.8.1.25}$$

Thus, the transformations

$$x \;=\; \frac{t}{1-t} \quad \textit{and} \quad t \;=\; \frac{x}{1+x} \tag{5.8.1.26}$$

represent the real root mapping required from the pseudo Bernstein and power forms, respectively. Figures 5.4 and 5.5 illustrate these mappings over their respective unit intervals.

Complex roots can be mapped by substituting for x and t the quantities

$$z \;=\; (c + i\,d) \quad \textit{and} \quad s \;=\; (u + i\,v) \tag{5.8.1.27}$$

where $i = \sqrt{-1}$. The mapping of the real and complex parts of the complex roots found from the pseudo Bernstein and power coefficients are represented by

$$c \;=\; \frac{u(1-u) - v^2}{(1-u)^2 + v^2}, \qquad d \;=\; \frac{v}{(1-u)^2 + v^2} \tag{5.8.1.28}$$

and

$$u \;=\; \frac{c(1+c) - d^2}{(1+c)^2 + d^2}, \qquad v \;=\; \frac{d}{(1+c)^2 + d^2}, \tag{5.8.1.29}$$

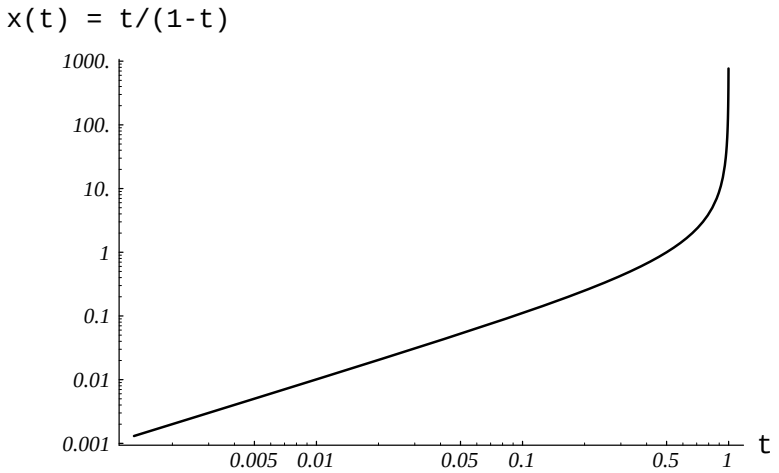

Figure 5.4: Log-log plot of pseudo-mapping $x(t) = t/(1-t)$ over the region $t \in [0, 1]$.

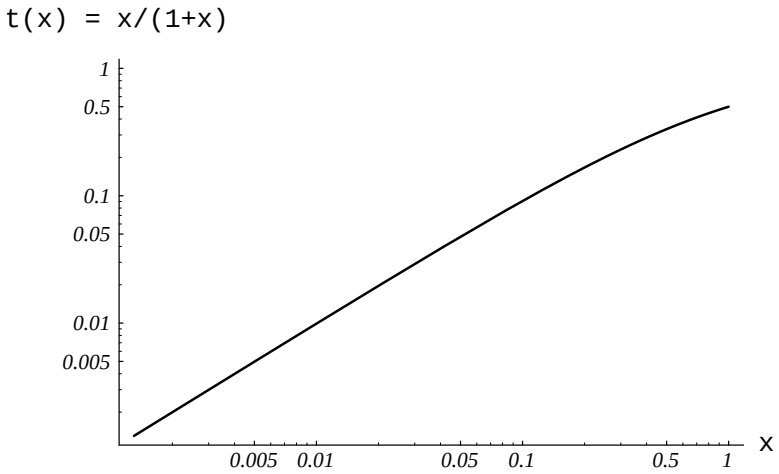

Figure 5.5: Log-log plot of pseudo-mapping $t(x) = x/(1+x)$ over the region $x \in [0, 1]$.

respectively. Note that when the complex parts $d = 0$ and $v = 0$, equations (5.8.1.28) and (5.8.1.29) simplify to

$$c \;=\; \frac{u}{(1-u)} \qquad \textit{and} \qquad u \;=\; \frac{c}{(1+c)} \tag{5.8.1.30}$$

which is equivalent to equations (5.8.1.26).

5.8.2 Algorithms: ConvCoefs$_{b2\hat{p}}$ and ConvCoefs$_{p2\hat{b}}$

Algorithm 5.8.2.1 (ConvCoefs$_{b2\hat{p}}$ $(\{b_i,\ \hat{p}_i\}_{i=0}^{n})$) *Convert Bernstein coefficients* $\{b_i\}$ *to pseudo-power coefficients* $\{\hat{p}_i\}$.

1. $\hat{s}_0 \;=\; b_0$
2. $\{\ \hat{s}_i \;=\; b_i\ \binom{n}{i}\ \}_{i=1}^{n-1}$
3. $\hat{s}_n \;=\; b_n$

End of ConvCoefs$_{b2\hat{p}}$

Algorithm 5.8.2.2 (ConvCoefs$_{p2\hat{b}}$ $(\{p_i,\ \hat{b}_i\}_{i=0}^{n})$) *Convert power coefficients* $\{p_i\}$ *to pseudo-Bernstein coefficients* $\{\hat{b}_i\}$.

1. $\hat{b}_0 \;=\; s_0$
2. $\{\ \hat{b}_i \;=\; s_i\ /\ \binom{n}{i}\ \}_{i=1}^{n-1}$
3. $\hat{b}_n \;=\; s_n$

End of ConvCoefs$_{p2\hat{b}}$

5.8.3 Algorithms: ConvRoots$_{\hat{b}}$ and ConvRoots$_{\hat{p}}$

Algorithm 5.8.3.1 (ConvRoots$_{\hat{b}}$ $(\{r_{\Re_i},\ r_{\Im_i}\}_{i=1}^{n_{roots}})$) *The initial computed pseudo-Bernstein form real and complex roots* $\{r_{\Re_i}, r_{\Im_i}\}$ *are converted and replaced by their power form roots.*

1. For $(i = 1, 2, \ldots, n_{roots})$
2. $\quad s \;=\; 1 \;-\; r_{\Re_i}$
3. $\quad sq \;=\; r_{\Im_i}^2$
4. $\quad d \;=\; s^2 \;+\; sq$
5. $\quad r_{\Re_i} \;=\; (r_{\Re_i}\ s \;-\; sq)\ /\ d$
6. $\quad r_{\Im_i} \;=\; r_{\Im_i}\ /\ d$

   EndFor$_i$

End of ConvRoots$_{\hat{b}}$

If only real roots are needed then replace steps 2-6 with

$$r_{\Re_i} \;=\; \frac{r_{\Re_i}}{1 \;-\; r_{\Re_i}}.$$

Algorithm 5.8.3.2 ($\textsc{ConvRoots}_{\hat{p}}$ $(\{r_{\Re_i},\ r_{\Im_i}\}_{i=1}^{n_{roots}})$) *The initial computed pseudo-power form real and complex roots* $\{r_{\Re_i}, r_{\Im_i}\}$ *are converted and replaced by their Bernstein form roots.*

1. For $(i = 1, 2, \ldots, n_{roots})$
2. $\quad s \;=\; 1 \;+\; r_{\Re_i}$
3. $\quad sq \;=\; r_{\Im_i}^2$
4. $\quad d \;=\; s^2 \;+\; sq$
5. $\quad r_{\Re_i} \;=\; (r_{\Re_i}\ s \;+\; sq)\ /\ d$
6. $\quad r_{\Im_i} \;=\; r_{\Im_i}\ /\ d$

   EndFor$_i$

End of $\textsc{ConvRoots}_{\hat{p}}$

If only real roots are needed, then replace steps 2-6 with

$$r_{\Re_i} \;=\; \frac{r_{\Re_i}}{1 \;+\; r_{\Re_i}}.$$

Note that roots only need to be remapped when they are found from pseudo converted coefficients, as opposed to other methods presented in [FR88, LR81, Riv70, CS66] and [§ 3.4].

5.9 Closed Form Real Root Solvers

This section presents the implementation of closed–form solvers [§ 2.3.1] for quadratic, cubic, and quartic polynomials in power form modified to solve for only real roots. These are called $\textsc{Psolver}_2$, $\textsc{Psolver}_3$, and $\textsc{Psolver}_4$, respectively. The algorithms consider degenerate polynomial cases, as well as applying linearized running error analysis [§ 3.2.2] to their discriminant formations to assist in detecting and estimating multiple root cases. $\textsc{Psolver}_2$ integrates numerical considerations from algorithms found in [Lod90, PFTV90, Sch90a].

PSOLVER$_3$ and PSOLVER$_4$ primarily follow the considerations outlined in [Lod90] taking into consideration implementations outlined in [PFTV90, Sch90a] for the cubic and [Sch90a] for the quartic real root solvers. Valuable insights regarding numerical considerations for solving real roots from closed–form cubic equations are presented in [Vig78]; unfortunately this algorithm was not completely implemented for comparison.

We first make some preliminary remarks pertaining to closed–form solutions for quadratic, cubic, and quartic polynomial equations represented in power form $p(x) = \sum_{i=0}^{n} p_i x^i = 0$, indicating real root strategies along with numerical considerations pertaining to discriminant error analysis.

Quadratic Solution

The solution of the quadratic polynomial equation

$$p_2 x^2 + p_1 x + p_0 = 0 \tag{5.9.0.31}$$

transformed into monic form

$$x^2 + q_1 x + q_0 = 0 \qquad (q_2 = 1) \tag{5.9.0.32}$$

is represented by

$$x_{1,2} \;=\; \frac{-q_1 \pm \sqrt{q_1^2 - 4q_0}}{2} \tag{5.9.0.33}$$

with the discriminant

$$\delta = q_1^2 - 4q_0 \tag{5.9.0.34}$$

yielding a corresponding computed bounding error of

$$\varepsilon_\delta \;\leq\; \left(\; |\delta| \;+\; q_1^2 \; \right) \; \eta \;+\; \left(\; 2 \; q_1^2 \; \left(\frac{\varepsilon_{q_1}}{q_1} \right) \;+\; 4 \; q_0 \; \left(\frac{\varepsilon_{q_0}}{q_0} \right) \; \right), \tag{5.9.0.35}$$

where ε_δ is computed by applying linearized running error analysis to δ, and

$$\left(\frac{\varepsilon_{q_1}}{q_1} \right) = \eta + \left(\frac{\varepsilon_{p_1}}{p_1} \right) + \left(\frac{\varepsilon_{p_2}}{p_2} \right), \qquad \left(\frac{\varepsilon_{q_0}}{q_0} \right) = \eta + \left(\frac{\varepsilon_{p_0}}{p_0} \right) + \left(\frac{\varepsilon_{p_2}}{p_2} \right) \tag{5.9.0.36}$$

are relative error bounds taking into consideration the monic transformation of equation (5.9.0.32) and the inherent error due to the coefficients $\{p_i\}$. An alternate form of the solution (5.9.0.33) is given by

$$x_{1,2} \;=\; \frac{-q_1}{2} \;\pm\; \sqrt{\left(\frac{q_1}{2}\right)^2 - q_0} \tag{5.9.0.37}$$

with the discriminant [Sch90a]

$$\delta = \left(\frac{q_1}{2}\right)^2 - \; q_0 \tag{5.9.0.38}$$

yielding the computed discriminant error of

$$\varepsilon_\delta \;\leq\; \left(\;|\delta| \;+\; q_1^2 \;\right) \;\eta \;+\; \left(\;2\; q_1^2 \;\left(\frac{\varepsilon_{q_1}}{q_1}\right) \;+\; q_0 \;\left(\frac{\varepsilon_{q_0}}{q_0}\right) \;\right), \tag{5.9.0.39}$$

where the inherent error applied to the q_0 term is now reduced by a factor of 4 compared to equation (5.9.0.35), thus minimizing the effect of any relative error incurred from $\left(\frac{\varepsilon_{q_0}}{q_0}\right)$ (see considerations below).

Depending on whether the discriminant δ is $<$, $=$, or $>$ 0 there are 0, 2 identical, or 2 distinct real roots, respectively. Numerical accuracy is enhanced in computing 2 identical real roots by substituting the condition $(\delta \leq \varepsilon_\delta)$ for the strict condition $(\delta > 0)$, and for computing 2 distinct real roots by replacing the inherent subtractive cancellation due to the radical term in the equation (5.9.0.37) with a reducible subtraction operation [Vig78, Lod90, PFTV90] that calculates the first root according to the sign of q_1 by

$$r_1 \;=\; -\left(\;\frac{q_1}{2} \;+\; \textsc{Sign}\left(\frac{q_1}{2}\right) \;\sqrt{|\delta|} \;\right) \tag{5.9.0.40}$$

where the second root is calculated by $r_2 = \frac{q_0}{r_1}$.

Cubic and Quartic Solutions

Considerations similar to those described above apply also to the closed–form treatments of cubic and quartic equations. The particulars are readily developed from the algorithm descriptions given below; in the interest of brevity we omit a detailed discussion here.

5.9.1 Algorithms: PSOLVER$_2$, PSOLVER$_3$, and PSOLVER$_4$

Algorithm 5.9.1.1 (PSOLVER$_2$ $(t,\ \{p_i\}_{i=0}^{2},\ \{r_j\}_{j=1}^{n_{roots}})$) *Standard closed–form real root quadric solver returns $n_{roots} = 0, 2$ roots $\{r_j\} \in \Re$.*

□ TEST degenerate polynomial cases.

1. IF (ISCOEF0 (p_2))
2. IF (ISCOEF0 (p_1)) $n_{roots} = 0$
3. ELSE RECORDROOTS (1 @ $\frac{-p_0}{p_1} \rightarrow \{r_j\}$)
4. RETURN

ENDIF$_{p_2}$

□ CONVERT coefficients to normal form.

5. $q_1 = \frac{p_1}{2\ p_2}$
6. $q_0 = \frac{p_0}{p_2}$

□ COMPUTE the discriminant δ and respection error ε_δ.

7. $\delta = q_1^2 - q_0$
8. $\varepsilon_\delta = (|\delta| + 7\ q_1^2 + 3\ |q_0|)\ \eta$

□ CASE: 0 roots in $\Re$ — $(\delta < 0)$.

9. $n_{roots} = 0$

□ CASE: 2 identical roots in $\Re$ — $(\delta = 0)$.

10. IF ($|\delta| \leq \varepsilon_\delta$)
11. RECORDROOTS (2 @ $-q_1 \rightarrow \{r_j\}$)

□ CASE: 2 distinct roots in $\Re$ — $(\delta > 0)$.

12. ELSEIF $(\delta > -\varepsilon_\delta)$
13. RECORDROOTS (1 @ $-(q_1 + \text{SIGN}(q_1)\ \sqrt{|\delta|}) \rightarrow \{r_j\}$)
14. RECORDROOTS (1 @ $\frac{q_0}{r_1} \rightarrow \{r_j\}$)

ENDIF$_\delta$

END OF PSOLVER$_2$

The above algorithm makes use of the following functions:

ISCOEF0 (c): TEST if coefficient c approximates zero

SIGN(b): Return the sign of b

SORT$_{asc}(\{v_i\})$: Sort values $\{v_i\}$ in ascending order

Algorithm 5.9.1.2 (PSOLVER$_3$ $(t,\ \{p_i\}_{i=0}^{3},\ \{r_j\}_{j=1}^{n_{roots}})$) *Standard closed–form real root cubic solver returns n_{roots} roots $\{r_j\} \in \Re$.*

□ CASE: degenerate cubic ($p_3 \approx 0$)

1. IF (IsCoef0 (p_3))
2. PSOLVER$_2$ ($\{p_i\}_{i=0}^{2}$, $\{r_j\}_{j=1}^{n_{roots}}$)
3. RETURN

□ CASE: other zero coefficients

4. ELSE

□ CASE: 1 $root = p_0 \approx 0$

5. IF (IsCoef0 (p_0))
6. RecordRoots ($1 \ @ \ x = 0 \ \rightarrow \ \{r_j\}$)

□ CASE: 2 $roots = p_1 \approx p_0 \approx 0$

7. IF (IsCoef0 (p_1))
8. RecordRoots ($1 \ @ \ x = 0 \ \rightarrow \ \{r_j\}$)

□ CASE: 3 $roots = p_2 \approx p_1 \approx p_0 \approx 0$

9. IF (IsCoef0 (p_2))
10. RecordRoots ($1 \ @ \ x = 0 \ \rightarrow \ \{r_j\}$)

□ CASE: 2 $roots = 0$ & single root in $\Re$

11. ELSE
12. RecordRoots ($1 \ @ \ \frac{-p_2}{p_3} \ \rightarrow \ \{r_j\}$)

ENDIF$_{p_2 \approx 0}$

□ CASE: single & quadric roots in $\Re$

13. ELSE
14. PSOLVER$_2$ ($\{p_i\}_{i=1}^{3}$, $\{r_j\}_{j=2}^{n_{roots}}$)

ENDIF$_{p_1 \approx 0}$

15. RETURN

ENDIF$_{p_0 \approx 0}$

ENDIF$_{p_3 \approx 0}$

□ CONVERT to monic form ($q(x) = p(x) \ / \ p_3$)

16. $\{q_2, \ q_1, \ q_0\} = \{\frac{p_2}{p_3}, \ \frac{p_1}{p_3}, \ \frac{p_0}{p_3}\}$

□ CHANGE of variable: $x = y - \frac{q_2}{3}$

□ SOLVE cubic using reduced/canonical cubic

□ $y^3 + (\ q_1 - \frac{1}{3} q_2^2 \ ) y + (\ q_0 - \frac{1}{3} q_2 q_1 + \frac{2}{27} q_2^3 \ ) = 0$

17. $\hat{p}_1 = q_1 - \frac{1}{3} \ q_2^2$
18. $\varepsilon_{\hat{p}_1} \ = \ (\ |\hat{p}_1| \ + \ |q_1| + \ \frac{4}{3} \ q_2^2 \ ) \ \eta$
19. $\hat{p}_0 \ = \ q_0 \ - \ \frac{1}{3} \ q_2 q_1 \ + \ \frac{2}{27} \ q_2^3$
20. $\varepsilon_{\hat{p}_0} = (\ |\hat{p}_0| \ + \ \frac{1}{2} \ |q_0| \ + \ |q_2| \ (\ \frac{7}{27} \ q_2^2 \ + \ \frac{1}{2} \ |q_1| \ ) \ ) \ \eta$

□ COMPUTE discriminant δ

21. $\delta \ = \ \frac{4}{27} \ \hat{p}_1^3 + \hat{p}_0^2$
22. $\varepsilon_\delta = (\ |\delta| + \frac{16}{27} \ |\hat{p}_1^3| \ + \ \hat{p}_0^2 \ ) \ \eta \ + \ (\ \frac{12}{27} \ \hat{p}_1^2 \ \varepsilon_{\hat{p}_1} \ + \ 2 \ |\hat{p}_0| \ \varepsilon_{\hat{p}_0} \ )$

□ CASE: 1 root in $\Re$ ($\delta > 0$)

23. IF ($\delta > \varepsilon_\delta$)
24. $w = \sqrt{\delta}$
25. $\alpha = \frac{w + \hat{p}_0}{2}$
26. $\beta = \frac{w - \hat{p}_0}{2}$
27. IF ($\hat{p}_0 < 0$)
28. RECORDROOTS (1 @ ($\frac{\beta}{3} - \frac{1}{3}\,\hat{p}_1\,\beta^{\frac{-1}{3}} - \frac{q_2}{3}$) $\rightarrow$ $\{r_j\}$)
29. ELSE
30. RECORDROOTS (1 @ ($-\frac{\alpha}{3} + \frac{1}{3}\,\hat{p}_1\,\alpha^{\frac{-1}{3}} - \frac{q_2}{3}$) $\rightarrow$ $\{r_j\}$)

ENDIF$_{\hat{p}_0 < 0}$

□ CASE: 3 roots in $\Re$ ($\delta \leq 0$)

31. ELSE

□ CASE: cubic root in $\Re$ ($\delta \approx 0,\ p \approx 0$)

32. IF ($|\delta| \leq \varepsilon_\delta$ AND $|\hat{p}_1| \leq \varepsilon_{\hat{p}_1}$)
33. RECORDROOTS (3 @ $\frac{-q_2}{3} \rightarrow \{r_j\}$)

□ CASE: single & double roots in $\Re$ ($\delta \approx 0,\ \hat{p}_1 < 0$)

□ AND CASE: 3 distinct roots in $\Re$ ($\delta < 0$)

34. ELSE
35. $\rho = \sqrt{-\frac{1}{3}\hat{p}_1}$

□ SUGGESTION 1 — not yet implemented

36. $t = -\frac{1}{2}\hat{p}_0 \;/\; \rho^3$

□ SUGGESTION 2 — not yet implemented

□ AVOID round-off problems — $cos^{-1}(-1 \leq t \leq 1)$

37. IF ($t > 1$) $t = 1$
38. ELSEIF ($t > 1$)$Int = -1$
39. $\theta = \frac{cos^{-1}(t)}{3}$
40. RECORDROOTS (1 @ ($-\frac{q_2}{3} + \rho\, 2\, cos(\theta)$) $\rightarrow$ $\{r_j\}$)
41. RECORDROOTS (2 @ ($-\frac{q_2}{3} - \rho\,(cos(\theta) \mp \sqrt{3}\, sin(\theta))$) $\rightarrow$ $\{r_j\}$)

ENDIF$_{\delta \leq \varepsilon_\delta}$

ENDIF$_{\delta > \varepsilon_\delta}$

END OF PSOLVER$_3$

Algorithm 5.9.1.3 (PSOLVER$_4$ (t, $\{p_i\}_{i=0}^{4}$, $\{r_j\}_{j=1}^{n_{roots}}$)) *Standard closed–form real root quartic solver returns n_{roots} roots $\{r_j\} \in \Re$.*

□ CASE: degenerate quartic case $p_4 \approx 0$

1. IF (ISCOEF0 (p_4))
2. PSOLVER$_3$ ($\{p_i\}_{i=0}^{3}$, $\{r_j\}_{j=1}^{n_{roots}}$)

3. RETURN
   □ CASE: other zero coefficient cases.
4. ELSE
   □ CASE: 1 $root = p_0 \approx 0$
5. IF (ISCOEF0 (p_0))
6. RECORDROOTS (1 @ $x = 0 \rightarrow \{r_j\}$)
   □ CASE: 2 $roots = p_1 \approx p_0 \approx 0$
7. IF (ISCOEF0 (p_1))
8. RECORDROOTS (1 @ $x = 0 \rightarrow \{r_j\}$)
   □ CASE: 3 $roots = p_2 \approx p_1 \approx p_0 \approx 0$
9. IF (ISCOEF0 (p_2))
10. RECORDROOTS (1 @ $x = 0 \rightarrow \{r_j\}$)
    □ CASE: 4 $roots = p_3 \approx p_2 \approx p_1 \approx p_0 \approx 0$
11. IF (ISCOEF0 (p_3))
12. RECORDROOTS (1 @ $x = 0 \rightarrow \{r_j\}$)
13. ELSE
14. RECORDROOTS (1 @ $\frac{-p_3}{p_4} \rightarrow \{r_j\}$)
    ENDIF$_{p_3 \approx 0}$
    □ CASE I: $2 roots = p_1 \approx p_0 \approx 0$ & quadric roots in $\Re$.
15. ELSE
16. PSOLVER$_2$ ($\{p_i\}_{i=2}^{4}$, $\{r_j\}_{j=3}^{n_{roots}}$)
    ENDIF$_{p_2 \approx 0}$
    □ CASE: 1 $root = p_0 \approx 0$ & cubic roots in $\Re$
17. ELSE
18. PSOLVER$_3$ ($\{p_i\}_{i=1}^{4}$, $\{r_j\}_{j=2}^{n_{roots}}$)
    ENDIF$_{p_1 \approx 0}$
19. RETURN
    ENDIF$_{p_0 \approx 0}$
    ENDIF$_{p_4 \approx 0}$
    □ CASE II: $p_3 \approx p_1 \approx 0$ $Roots = \pm\sqrt{quadric\ roots\ in\ \Re}$
20. IF (ISCOEF0 (p_3) AND ISCOEF0 (p_1))
21. $\{\hat{p}_2, \hat{p}_1, \hat{p}_0\} = \{p_4, p_2, p_0\}$
22. PSOLVER$_2$ ($\{\hat{p}_i\}_{i=0}^{2}$, $\{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}}$)
23. { RECORDROOTS (2 @ $\pm\sqrt{\hat{r}_j} \rightarrow \{r_j\}$) }$_{i=1}^{\hat{n}_{roots}}$
24. RETURN
    ENDIF$_{p_3 \approx\ p_1 \approx 0}$
    □ CONVERT to monic form ($q(x) = p(x)/p_4$)
25. $\{q_i = \frac{p_i}{p_4}\}_{i=0}^{3}$

□ PREPARE special quartic method

26. $\gamma_0 \;=\; q_2 \;-\; \frac{1}{4}\, q_3^2$
27. $\gamma_1 \;=\; q_3\, \gamma_0 \;-\; 2\, q_1$
28. $\varepsilon_{\gamma_1} \;=\; (\; |\gamma_1| \;+\; |\gamma_0|\, (2|q_3| + 3) + 6|q_1| + |q_3|\, (\frac{7}{4} q_3^2 + 3|q_2|)\;)\; \eta$
29. $\gamma_2 \;=\; q_0 \;+\; q_3^2 \;-\; q_1^2$
30. $\varepsilon_{\gamma_2} \;=\; (\; |\gamma_2| \;+\; 10|q_0|\, q_3^2 \;+\; 7 q_1^2 \;)\; \eta$

□ CASE: special quartic method

31. IF $(\; |\gamma_1| \le \varepsilon_{\gamma_1} \;)$
32. $\{\hat{p}_2, \hat{p}_1\} \;=\; \{1, \frac{q_3}{2}\}$
33. $\beta \;=\; \frac{q_1}{q_3}$

□ CASE III: 2 double roots in $\Re$ $(gamma_1 \;\approx\; \gamma_2 \;\approx\; 0)$

34. IF $(\; |\gamma_2| \le \varepsilon_{\gamma_2} \;)$
35. $\hat{p}_0 \;=\; \beta$
36. PSOLVER$_2$ $(\; \{\hat{p}_i\}_{i=0}^{2},\; \{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}} \;)$
37. $\{$ RECORDROOTS $(\; 2 \;@\; \hat{r}_j \;\rightarrow\; \{r_j\} \;)\; \}_{j=1}^{\hat{n}_{roots}}$
38. RETURN

□ CASE IV: $(\gamma_1 \approx 0,\; \gamma_2 < 0)$

39. ELSEIF $(\; \gamma_2 < 0 \;)$
40. $\rho = \sqrt{\beta^2 - q_0}$

□ SOLVE quartic factor: $x^2 + \frac{q_3}{2}\, x + (\beta + \rho)$

41. $\hat{p}_0 = \beta + \rho$
42. PSOLVER$_2$ $(\; \{\hat{p}_i\}_{i=0}^{2},\; \{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}} \;)$
43. $\{$ RECORDROOTS $(\; 1 \;@\; \hat{r}_j \;\rightarrow\; \{r_j\} \;)\; \}_{j=1}^{\hat{n}_{roots}}$

□ SOLVE quartic factor: $x^2 + \frac{q_3}{2}\, x + (\beta - \rho)$

44. $\hat{p}_0 = \beta - \rho$
45. PSOLVER$_2$ $(\; \{\hat{p}_i\}_{i=0}^{2},\; \{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}} \;)$
46. $\{$ RECORDROOTS $(\; 1 \;@\; \hat{r}_j \;\rightarrow\; \{r_j\} \;)\; \}_{j=1}^{\hat{n}_{roots}}$
47. RETURN

ENDIF$_{\gamma_2}$

ENDIF$_{\gamma_1}$

□ SOLVE quartic using general method of reduced cubic

□ $y^3 + 2q_2 y^2 + (q_2^2 + q_3 q_1 - 4q_0)\; y + (q_1^2 - q_3 q_2 q_1 + q_3^2 q_0) = 0$

48. $\{\hat{p}_3, \hat{p}_2, \hat{p}_1, \hat{p}_0\} = \{1,\; -2q_2,\; q_2^2 + q_3 q_1 - 4q_0,\; q_1^2 - q_3 q_2 q_1 + q_3^2 q_0\}$

□ SOLVE for real roots of reduced cubic

49. PSOLVER$_3$ $(\; \{\hat{p}_i\}_{i=0}^{3},\; \{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}}\} \;)$
50. $u = \parallel (q_2^2 - \hat{r}_j)^2 - 4q_0 \parallel$
51. $v = \parallel q_3^2 - 4\hat{r}_j \parallel$

□ METHOD I: $u > v$, where $\hat{r}_k$ is the root maximized u

52. IF ($u > v$)
53. $u = \sqrt{u}$
54. $w = q_2 - \hat{r}_k$
55. IF ($y < 0$)
56. $h_2 = \frac{w-z}{2}$
57. $h_1 = \frac{q_0}{h_2}$
58. ELSE
59. $h_1 = \frac{w+u}{2}$
60. $h_2 = \frac{q_0}{h_1}$

ENDIF$_{cy}$

61. $z = q_3\ y - 2q_1$
62. IF ($q_3\ z < 0$)
63. $g_2 = (q_3 - \frac{z}{u})\ /\ 2$
64. $g_1 = q_0\ \hat{r}_k\ /\ g_2$
65. ELSE
66. $g_1 = (q_3 + \frac{z}{u})\ /\ 2$
67. $g_2 = q_0\ \hat{r}_k\ /\ g_1$

ENDIF$_{q_3 z}$

□ METHOD II: $v > u$, where $\hat{r}_l$ is the root that maximized v

68. ELSE
69. $v = \sqrt{v}$
70. $y = q_2 - \hat{r}_l$
71. $z = q_3\ y - 2q_1$
72. IF ($y\ z < 0$)
73. $h2 = (y - \frac{z}{v})\ /\ 2$
74. $h_1 = q_0\ /\ h_2$
75. ELSE
76. $h_1 = (y + \frac{z}{v})\ /\ 2$
77. $h_2 = q_0\ /\ h_1$

ENDIF$_{yz}$

78. IF ($q_3 < 0$)
79. $g_2 = (q_3 - v)\ /\ 2$
80. $g_1 = \hat{r}_l\ /\ g_2$
81. ELSE
82. $g_1 = (q_3 + v)\ /\ 2$
83. $g_2 = \hat{r}_l\ /\ g_1$
84. ENDIF$_{q_3}$

ENDIF$_u$

□ SOLVE: $x^2 + g_1 x + h_1 = 0$

85. $\{\hat{p}_2, \hat{p}_1, \hat{p}_0\} = \{1, g_1, h_1\}$

86. PSOLVER$_2$ ($\{\hat{p}\}_{i=0}^{2}$, $\{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}}$)

87. { RECORDROOTS (1 @ $\hat{r}_j \rightarrow \{r_j\}$) }$_{j=1}^{\hat{n}_{roots}}$

□ SOLVE: $x^2 + g_2 x + h_2 = 0$

88. $\{\hat{p}_2, \hat{p}_1, \hat{p}_0\} = \{1, g_2, h_2\}$

89. PSOLVER$_2$ ($\{\hat{p}_i\}_{i=0}^{2}$, $\{\hat{r}_j\}_{j=1}^{\hat{n}_{roots}}$)

90. { RECORDROOTS (1 @ $\hat{r}_j \rightarrow \{r_j\}$) }$_{j=1}^{\hat{n}_{roots}}$)

END OF PSOLVER$_4$

Chapter 6

Polynomial Real Root-Finding Algorithms

This chapter presents several algorithms for finding all the real roots of a polynomial in Bernstein form expressed as an explicit Bézier curve, $\mathbf{P}_{[0,1]}$.

Section 6.1 presents an algorithm called BCHA$_1$ which applies the Bernstein convex hull property to create a sequence of approximating steps that converge to the leftmost real root. When the root is found, it is deflated and the march continues to the next root. Section [§ 6.2] extends the linear approximation step concept to quadratic, cubic, and quartic approximating steps which embody algorithms BCHA$_2$, BCHA$_3$, and BCHA$_4$.

The algorithms BCOM and BCHI in sections [§ 6.3.2.1] and [§ 6.4], respectively, are based on the isolate/refine strategy and involve different methods for root isolation. Once a root is isolated, each algorithm invokes a root refinement routine. Section [§ 6.3.2.1] applies the de Casteljau algorithm a few times and counts the sign changes in the resulting control polygons. If that fails to isolate the leftmost root, the leftmost root of the first derivative is computed. This procedure is applied recursively in the case of root clusters and multiple roots. The algorithm in section [§ 6.4] presents an isolation method wherein the t-axis is traversed in increments which are guaranteed to contain at most one root. Section [§ 6.5] presents the algorithm SIPS which investigates the use of the real roots from

isolator polynomials to isolate the real roots of a polynomial in power form of degree up to nine.

The numerical accuracy of these algorithms is enhanced by implementing the valuable insights on error tolerances and root conditioning provided by Farouki and Rajan [FR87, FR88], especially in dealing with multiple and clustered roots.

6.1 Bernstein Convex Hull Approximating Step (BCHA$_1$)

The *Bernstein convex hull approximating step algorithm* (BCHA$_1$) finds each real root on $[0,1]$ in ascending order by computing a sequence of approximating steps which are guaranteed to not skip over a real root. Each step is determined by exploiting the convex hull property [§ 2.1.2] 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 was first overviewed in [SP86], and fully discussed in an unpublished manuscript [SSd86]. Since then, it has appeared in [Gra89, Sch90a], and has been adapted to various applications and to higher dimensions under a technique referred to as Bézier clipping [SWZ89, Sed89, SN90a, SN90b].

6.1.1 Preliminary Concepts

BCHA$_1$ 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. The (linear) approximation step is outlined in algorithm BHULLAPPROX$_1$ below which finds the most positive (or negative) slope based on the left–most intersection of the convex hull with the t-axis.

This algorithm employs an effective *split* heurisitic which alleviates slow progression due to coefficients that vary by several orders of magnitude. The heuristic is set if the steepest

line is not defined by the line joining the first two coefficients of a Bernstein form polynomial, y_0 and y_1, whereupon the polynomial is subsequently subdivided. The algorithm works with the left segment only until all its roots are found, and then processes the right segment. The *split* heuristic works well because subdivision tends to even out the magnitudes of the coefficients.

Algorithm 6.1.1.1 (BHULLAPPROX$_1$ ($\{y_i\}_{i=0}^n$, *split*, t)) *Compute and return the left-most point* $t \in [0,1]$ *at which the* t*-axis intersects the convex hull of the control points of the explicit Bézier curve. Also, indicate possible slow convergence or multiple root conditions with the flag* split *which is set if the steepest slope is not a function of the first control point* y_1.

1. $split$ = FALSE
2. $slope$ = $y_1 - y_0$
   - □ Get most $+slope$
3. IF ($y_0 < 0$)
4. FOR ($j = 2, 3, \ldots, n$)
5. $slope_{tmp} = \frac{y_i - y_0}{i}$
6. IF ($slope_{tmp} > 0$ AND $slope_{tmp} > slope$)
7. $split$ = TRUE
8. $slope$ = $slope_{tmp}$

   ENDIF$_{slope_{tmp}}$

   ENDFOR$_i$
   - □ Get most $-slope$
9. ELSE
10. FOR ($j = 2, 3, \ldots, n$)
11. $slope_{tmp} = \frac{y_i - y_0}{i}$
12. IF ($slope_{tmp} < 0$ AND $slope_{tmp} < slope$)
13. $split$ = TRUE
14. $slope$ = $slope_{tmp}$

    ENDIF$_{slope_{tmp}}$

    ENDFOR$_i$

    ENDIF$_{y_0}$
15. $slope$ = n $slope$
    - □ Compute where hull's slope crosses abscissa
16. $t = -\frac{y_0}{slope}$

END OF BHULLAPPROX$_1$

6.1.2 Algorithm: BCHA$_1$

BCHA$_1$ is divided into two primary stages as depicted in the following algorithm outlines.

Algorithm 6.1.2.1 (BCHA$_1$ ($LeftRoot_{only}$, t_{tol}, $\{c_i, \varepsilon_{c_i}\}_{i=0}^{n}$, $\{r_j\}_{j=1}^{n_{roots}}$)) BCHA$_1$ *receives as input a leftmost–root–only flag* $LeftRoot_{only}$, *a solution tolerance* t_{tol}, *and Bernstein coefficients and their errors* $\{y_i, \varepsilon_{y_i}\}$ *defined over the domain* $[0, 1]$ *and returns* n_{roots} *roots* $\{r_j\}$.

BCHA$_1$.*1.* REAL ROOT PRE-APPROXIMATION STEPS, *and*

BCHA$_1$.*2.* REAL ROOT APPROXIMATION STEPS,

END OF BCHA$_1$

The respective steps pertaining to these stages are outlined below. The pre-approximation stage is a variation of similar stages presented in [§ 6.3.2] and [§ 6.4.2].

BCHA$_1$.1: REAL ROOT PRE-APPROXIMATION STEPS

1.1 ASSIGN a master (polynomial) segment structure.
(The master segment structure consists of the degree N, master polynomial coefficients and their error coefficients $\{ Y_i, E_i\}$, and the master domain limits fixed to the unit interval $[T_{min}, T_{max}] = [0, 1]$.)

1.2 NORMALIZE the master coefficients $\{ Y_i, E_i\}$.
(See [§ 5.4])

1.3 TEST for rightmost root(s) at the right interval limit T_{max}.
(Use: BEND0$_{right}$ in [§ 5.5])

1.4 INITIALIZE both a left and a right (local) segment structure to the master segment structure.
(The left (inner and active) coefficients are represented by $\{y_i^L, e_i^L\}$ with degree n^L, and the right (outer and pending) coefficients are represented by $\{y_i^R, e_i^R\}$ with degree n^R, and their respective domain limits in which they are defined are initially set to $[t_0^L, t_n^L] = [t_0^R, t_n^R] = [T_{min}, T_{max}]$.)

1.5 INITIALIZE the step progression flag $damn$ = FALSE.
(When the progression reaches a root, $damn$ is true.)

1.6 PROCEED TO STEP 2.1.

BCHA$_1$.2: REAL ROOT APPROXIMATION STEPS

2.1 TEST if progression is *damn*ed.
(IF (*damn*), then
BSUBDIV$_{right}$ segment $\{y_i^R, e_i^R\}$ at $t_k^R = \frac{t_0^L - t_0^R}{t_n^R - t_0^R}$,
RECORD the root at t_L, and
BDEFLATE$_{left}$ t_L from $\{y_i^L, e_i^L\}$,
PROCEED TO STEP 2.1.)

2.2 TEST if control polygon straddles t-axis.
(Check the number of sign variations in the coefficients, keeping tract of the 1st coefficient which changes sign denoted as i_{cross_1}.)

2.3 CASE for more than 1 sign change — more than 1 root.

2.3.1 COMPUTE an approximating step t_k^L. (The linear step is computed using BHULLAPPROX$_1$ which is the intersection of the convex hull with the t-axis, and the non-linear steps use BHULLAPPROX$_{2-4}$.)

2.3.2 STORE $\{y_i^L, e_i^L\}$ as 1st temp segment $\{y_i^{T_1}, e_i^{T_1}\}$ in case step over root.

2.3.3 SUBDIVIDE the right segment at t_k^L returning the left subdivided hull as a 2nd temporary segment, the right subdivide hull as the new left segment.
(BSUBDIV$_{right}$ segment $\{y_i^L, e_i^L\}$ at t_k^L returning $\{y_i^{T_2}, e_i^{T_2}\}$.

2.3.4 TEST for function convergence.
(IF ($y_0^L \leq e_0^L$), then *damn* = TRUE, and PROCEED TO STEP .2.1)

2.3.5 TEST for overshooting the root.
(IF ($y_0^{T_2} \leq e_0^{T_2}$ AND $y_0^L \leq e_0^L$), then
REFINE the isolated root using modified regula falsi,
RESTORE $\{y_i^L, e_i^L\} = \{y_i^{T_1}, e_i^{T_1}\}$,
SUBDIVIDE the left segment at the root, returning the right subdivided hull as the new left segment.)

2.3.6 TEST progress heuristic *split*; and if true,
SUBDIVIDE the left (inner) segment based on i_{cross_1},
UPDATING the left segment with the left subdivided hull.
(IF (*split*), then
BSUBDIV$_{left}$ segment $\{y_i^L, e_i^L\}$ at $t = i_{cross_1}/n^L$.

2.3.7 PROCEED TO STEP 2.1.

2.4 CASE control polygon does not straddle t-axis — a single real root.

2.4.1 TEST left limit for possible root.
(IF ($damn = y_0^L \leq e_0^L$) PROCEED TO STEP 2.1.)

2.4.2 TEST for more roots in the left (inner) segment.
(IF ($y_n^L < y_0^R$), then
BSUBDIV$_{right}$ segment $\{y_i^R, e_i^R\}$ at $t_k^R = \frac{t_n^L - t_0^R}{t_n^R - t_0^R}$,
ASSIGN $\{y_i^R, e_i^R\} = \{y_i^L, e_i^L\}$ and $t_0^L = t_0^R$, $t_n^L = t_n^R$.
PROCEED TO STEP 2.1.)

2.5 RECORD any right-most roots at T_{max} found from STEP 1.3, and RETURN.

A simplified example of the real root approximation stage for the leftmost root proceeds as illustrated in Figure 6.1, where each approximating step $t_k^L = t_k$ computed from BHULLAPPROX$_1$ progresses towards the root from the left. The progression is damned at the root when the left-most control point approximates its coefficient error bound which consequently accumulates during each subdivision and deflation - see algorithms BSUBDIV$_{left}$ (5.1.1.1) and BDEFLATE$_{left}$ (5.3.2.2). The process is repeated for each real root.

Analytically, the approximating step never crosses the left-most root due to convex hull containment. Numerically, the computational error induced by roundoff [§ 3.1] may allow the iterate to step over the root. To remedy this anomaly, STEP 2.3.5 is implemented to check the inner polynomial segment for an isolated root which is then approximated via the modified regula falsi algorithm. It should be stressed that a mere absolute error check of the iterating approximate step is not an adequate convergence check for this problem, because a polynomial exhibiting flatness in the neighborhood of this bracketed interval usually returns prior to actual convergence.

6.2 Higher Degree Approximating Steps (BCHA$_{2-4}$)

This section describes modifications to BCHA$_1$ which provide for higher order convergence. These modifications are consolidated into the algorithm BHULLAPPROX$_{2-4}$ by substituting them for algorithm BHULLAPPROX$_1$ (6.1.1.1) in STEP 2.3.1 of algorithm BCHA$_1$ (6.1.2.1).

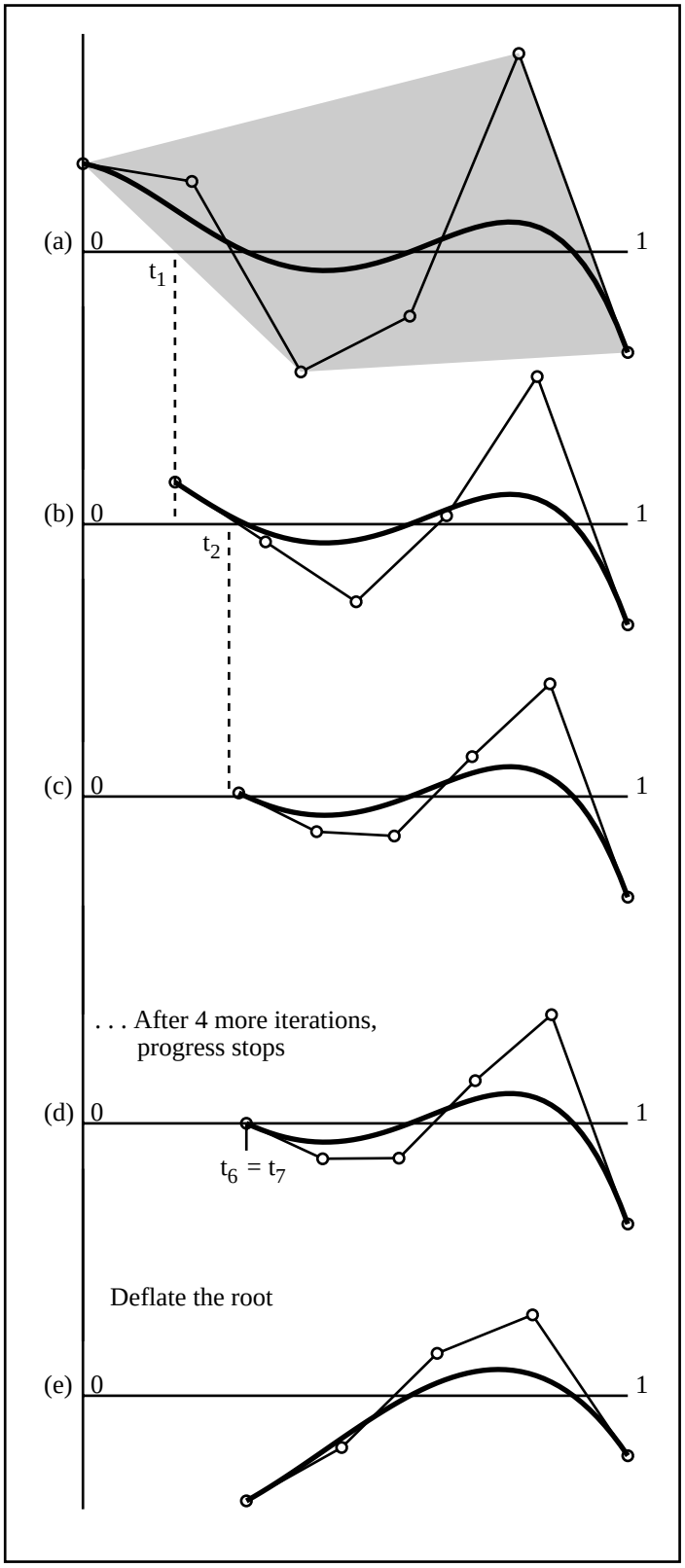

Figure 6.1: BCHA$_1$ root finding

6.2.1 Preliminary Concepts

BCHA$_1$ has the advantage that, unlike Newton's method, it will never step past a root. It has the disadvantage that a de Casteljau operation ($O(n^2)$) is required at each step, whereas Newton's method only requires function evaluation ($O(n)$).

A reasonable modification to BCHA$_1$ is to devise approximating polynomials of degrees two, three, and four whose leftmost roots are guaranteed to be less than the leftmost root of the given polynomial.

Such approximations are straightforward to construct using the explicit Bézier curve representation. Given $\mathbf{P}_{[a,b]}(t) = (t, y_{[a,b]}(t))$ and $\hat{\mathbf{P}}_{[a,b]}(t) = (t, \hat{y}_{[a,b]}(t))$ with with $y_{[a,b]}(a) = \hat{y}_{[a,b]}(a) < 0$ and $y_{[a,b]}(t) \leq \hat{y}_{[a,b]}(t)$ for $t \in [a,b]$, clearly the leftmost root of $\hat{\mathbf{P}}_{[a,b]}(t)$ will lie to the left of the leftmost root of $\mathbf{P}_{[a,b]}(t)$. If $\hat{\mathbf{P}}_{[a,b]}(t)$ is degree two, three, or four, its roots can quickly be found in closed form, and could provide an improved step for BCHA$_1$. Figure 6.2 shows examples of $\hat{\mathbf{P}}_{[a,b]}(t)$ of various degrees.

Of course, convergence is enhanced if $\hat{\mathbf{P}}_{[a,b]}(t)$ matches as many derivatives as possible with $\mathbf{P}_{[a,b]}(t)$ at $t = a$. If $\mathbf{P}_{[a,b]}(t)$ is degree n and $\hat{\mathbf{P}}_{[a,b]}(t)$ is degree $\hat{n}$, it is possible to match $\hat{n} - 1$ derivatives:

$$\mathbf{P}_0 = \hat{\mathbf{P}}_0 \tag{6.2.1.1}$$

For $\hat{n} \geq 2$:

$$\begin{aligned} \mathbf{P}'_{[a,b]}(a) &= \hat{\mathbf{P}}'_{[a,b]}(a) \\ n(\mathbf{P}_1 - \mathbf{P}_0) &= \hat{n}(\hat{\mathbf{P}}_1 - \hat{\mathbf{P}}_0) \\ \hat{\mathbf{P}}_1 = n(\mathbf{P}_1 - \mathbf{P}_0)/\hat{n} + \mathbf{P}_0 & \end{aligned}$$

For $\hat{n} \geq 3$:

$$\begin{aligned} \mathbf{P}''_{[a,b]}(a) &= \hat{\mathbf{P}}''_{[a,b]}(a) \\ n(n-1)(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0) &= \hat{n}(\hat{n}-1)(\hat{\mathbf{P}}_2 - 2\hat{\mathbf{P}}_1 + \hat{\mathbf{P}}_0) \\ \hat{\mathbf{P}}_2 &= \frac{n(n-1)}{\hat{n}(\hat{n}-1)}(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0) + 2\hat{\mathbf{P}}_1 - \mathbf{P}_0 \end{aligned}$$

For $\hat{n} \geq 4$:

$$\mathbf{P}'''_{[a,b]}(a) = \hat{\mathbf{P}}'''_{[a,b]}(a)$$

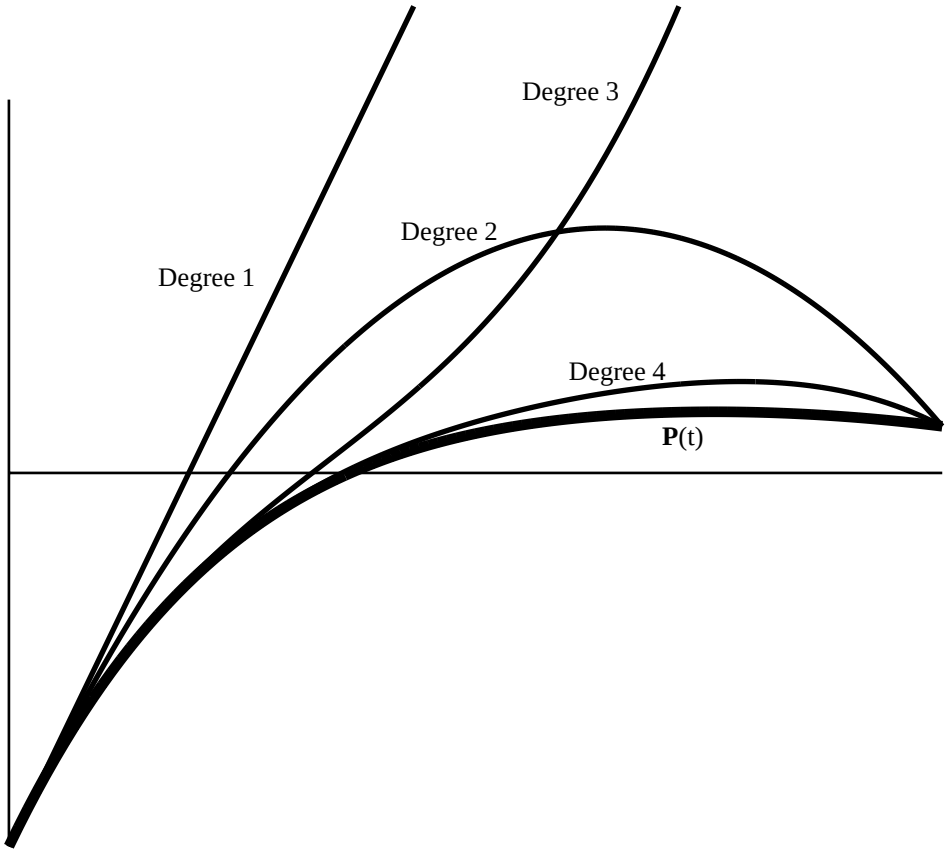

Figure 6.2: $\hat{\mathbf{P}}_{[a,b]}(t)$ of degrees one through four.

$$\begin{aligned} n(n-1)(n-2)(\mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0) &= \hat{n}(\hat{n}-1)(\hat{n}-2)(\hat{\mathbf{P}}_3 - 3\hat{\mathbf{P}}_2 + 3\hat{\mathbf{P}}_1 - \hat{\mathbf{P}}_0) \\ \hat{\mathbf{P}}_3 &= \frac{n(n-1)(n-2)}{\hat{n}(\hat{n}-1)(\hat{n}-2)}(\mathbf{P}_3 - 3\mathbf{P}_2 + 3\mathbf{P}_1 - \mathbf{P}_0) + 3\hat{\mathbf{P}}_2 - 3\hat{\mathbf{P}}_1 + p_0 \end{aligned}$$

This will fix the values of all but the last coefficient of $\hat{\mathbf{P}}_{[a,b]}(t)$, $\hat{\mathbf{P}}_{\hat{n}}$, which is chosen in a way that will assure $y_{[a,b]}(t) \leq \hat{y}_{[a,b]}(t)$. This is done by degree elevating $\hat{\mathbf{P}}_{[a,b]}(t)$ to degree n, and solving for $\hat{\mathbf{P}}_{\hat{n}}$ so that when $\hat{\mathbf{P}}_{[a,b]}(t)$, none of its control points lie beneath the respective control points of $\mathbf{P}_{[a,b]}(t)$.

Denote by $\mathbf{Q}(t)$ the degree elevation of $\hat{\mathbf{P}}(t)$. Because of the derivative constraints, we have

$$\mathbf{Q}_i \equiv \mathbf{P}_i, \quad i = 0, \ldots, \hat{n} - 1. \tag{6.2.1.2}$$

The remaining control points are functions of $\hat{\mathbf{P}}_{\hat{n}}$:

$$\mathbf{Q}_i = \frac{\sum_{\substack{j+k=i \\ j \in [0,\hat{n}]}} \binom{\hat{n}}{j}\binom{n-\hat{n}}{k}\hat{\mathbf{P}}_j}{\binom{n}{i}}; \quad i = \hat{n} \ldots n. \tag{6.2.1.3}$$

We now determine $\hat{\mathbf{P}}_{\hat{n}}$ which will force $\mathbf{Q}_i \geq \mathbf{P}_i$, $i = \hat{n} \ldots n$ (where $>$ denotes having a larger vertical coordinate):

$$\hat{\mathbf{P}}_{\hat{n}} \geq \frac{\binom{n}{i}}{\binom{n-\hat{n}}{k}} \left\lceil \mathbf{Q}_i - \sum_{\substack{j+k=i \\ j \in [0,\hat{n}-1]}} \frac{p_j(n1,k)(m,j)}{(n,i)} \right\rceil \qquad i = \hat{n} \ldots n \tag{6.2.1.4}$$

The following algorithm succinctly implements this polynomial approximation.

Algorithm 6.2.1.1 (BHULLAPPROX$_{2-4}$ $(\{y_i\}_{i=0}^{n}, \{\hat{y}_i\}_{i=0}^{\hat{n}})$) *Bernstein coefficients* $\{\hat{y}_i\}$ *are computed which optimally approximate any explicit Bernstein polynomial defined by* $\{y_i\}$ *over interval* $[a,b]$, *according to the specified degree* $\hat{n}$. *NOTE: The split heuristic is obtained by calling* BHULLAPPROX$_1$ *with the original coefficients — separate from this function.*

1. $\hat{y}_n = y_0$
2. $\hat{y}_1 = y_0 + \frac{n}{\hat{n}}(y_1 - y_0)$
3. IF $(\hat{n} > 2)$ $\hat{y}_2 = \frac{n(n-1)}{\hat{n}(\hat{n}-1)}(y_2 + 2y_1 - y_0) + 2y_1 - y_0$

4. IF $(\hat{n} > 3)$ $\hat{y}_3 = \frac{n(n-1)(n-2)}{\hat{n}(\hat{n}-1)(\hat{n}-2)}(y_3 - 3y_2 + 3y_1 - y_0) + 3y_2 - 3y_1 + y_0$
5. $sum = 0$
6. $m = n - \hat{n}$
7. IF$(y_0 > 0)$
8. $y_{min} = +$DOUBLEMAXVALUE
9. FOR $(i = \hat{n}, \hat{n}+1, \ldots, n)$
10. FOR $(j = 0, 1, \ldots, \hat{n})$
11. FOR $(k = 0, 1, \ldots, m)$
12. IF $(j + k == i)$
13. IF $(j == \hat{n})$ $coe = \binom{m}{k} / \binom{n}{i}$
14. ELSE $sum = sum + \hat{y}_j \binom{\hat{n}}{j} \binom{m}{k} / \binom{n}{i}$
ENDIF$_{j+k}$
ENDFOR$_k$
ENDFOR$_j$
15. $\hat{y}_{\hat{n}} = \frac{y_i - sum}{coe}$
16. IF $(\hat{y}_{\hat{n}} < y_{min})$ $y_{min} = \hat{y}_{\hat{n}}$
17. $sum = 0$
18. $\hat{y}_{\hat{n}} = y_{min}$
ENDFOR$_i$
19. ELSE
20. $y_{max} = -$DOUBLEMAXVALUE
21. FOR $(i = \hat{n}, \hat{n}+1, \ldots, n)$
22. FOR $(j = 0, 1, \ldots, \hat{n})$
23. FOR $(k = 0, 1, \ldots, m)$
24. IF $(j + k == i)$
25. IF $(j == \hat{n})$ $coe = \binom{m}{k} / \binom{n}{i}$
26. ELSE $sum = sum + \hat{y}_j \binom{\hat{n}}{j} \binom{m}{k} / \binom{n}{i}$
ENDIF$_{j+k}$
ENDFOR$_k$
ENDFOR$_j$
27. $\hat{y}_{\hat{n}} = \frac{y_i - sum}{coe}$
28. IF $(\hat{y}_{\hat{n}} < y_{max})$ $y_{max} = \hat{y}_{\hat{n}}$
29. $sum = 0$
30. $\hat{y}_{\hat{n}} = y_{max}$
ENDFOR$_i$
ENDIF

END OF BHULLAPPROX$_{2-4}$

6.2.2 Higher Degree Approximating Step Considerations

The minimum root of the approximated polynomial $\hat{\mathbf{P}}_{[a,b]}$ obtained from the closed form real root solvers [§ 5.9] over the specified interval is used as the next approximation step for algorithms BCHA$_{2-4}$.

This application appears to be limited by numerical round-off error induced from the numerical closed form solution which permits the approximating step to nudge over a root. This is especially evident in the quartic version. Therefore STEP 2.3.5 in BCHA$_1$ handles this condition and checks whether the polynomial crosses the t-axis, and if so, refine the isolated root using a root approximation algorithm.

6.3 Bernstein Combined Subdivide & Derivative (BCOM)

The *Bernstein combined subdivide and derivative algorithm* (BCOM) is an isolate/refine strategy. Isolation commences with a binary search, and if a root is not isolated within a few iterations, the leftmost root of the first derivative is computed for use as a possible root isolation value. Once a root is isolated, it is refined using the modified regula falsi method.

BCOM is somewhat similar to the Lane-Riesenfeld root finding algorithm [LR81]. However, BCOM is substantially faster because it is able to isolate roots in fewer subdivisions, and because its refinement stage uses an $O(n)$ polynomial evaluation algorithm rather than the $O(n^2)$ de Casteljau algorithm used by the Lane-Riesenfeld algorithm.

This section provides a serial implementation of BCOM. [Zun89] outlines a recursive implementation of a similar algorithm.

6.3.1 Preliminary Concepts

BCOM couples the stable condition inherent in Bernstein subdivision [§ 5.1] with real root isolation principles based on Rolle's theorem [§ 2.2.2] which utilize the various derivative

levels of the polynomial to isolate each real root of the Bernstein form polynomial over a specified interval. In addition, the use of scaled polynomial coefficients suggested in [FR88] increases the efficiency of the modified Horner's algorithm presented in [§ 5.2].

6.3.2 Algorithm: BCOM

The main functions of BCOM are summarized in the following algorithm and outlined below in pseudo code format.

Algorithm 6.3.2.1 (BCOM ($LeftRoot_{only}$, t_{tol}, $\{c_i, \varepsilon_{c_i}\}_{i=0}^{n}$, $\{r_j\}_{j=1}^{n_{roots}}$)) BCOM *receives as input a leftmost–root–only flag* $LeftRoot_{only}$, *a solution tolerance* t_{tol}, *and Bernstein coefficients and their errors* $\{y_i, \varepsilon_{y_i}\}$ *defined over the domain* $[0, 1]$ *and returns* n_{roots} *roots* $\{r_j\}$.

BCOM.*1.* REAL ROOT PRE-ISOLATION STEPS

This stage INITIALIZES variables, ASSIGNING and NORMALIZING the master polynomial structure, TESTING for roots at the end control points, ASSIGNING a local polynomial structure, and then PROCEEDS to phase BCOM.*2. (See details in stage* BCOM.*1 outline below.)*

BCOM.*2.* REAL ROOT ISOLATION LOOP STEPS

This stage uses a combination of subdivision and derivative heuristics to ISOLATE the real roots into distinct ascending ordered intervals, and PROCEEDS to phase BCOM.*3 to approximate the root whenever a single sign change is detected in a coefficient sequence. After all the real roots are found, all right end roots are RECORDED and the program is EXITED. (See details in stage* BCOM.*2 outline below.)*

BCOM.*3.* (MULTIPLE) REAL ROOT APPROXIMATION LOOP STEPS

This stage APPROXIMATES and RETURNS each isolated root to its appropriate multiplicity. (See details in stage BCOM.*3 outline below.)*

END OF BCOM

BCOM.1: REAL ROOT PRE-ISOLATION STEPS

1.1 □ INITIALIZE counters for subdivide stack, derivatives, & roots.

$S_{top} = 0$	□ top of subdivided poly segment stack $\{ S^i_{poly} \}_{i=1}^{S_{top}}$
$l = 0$	□ derivative level index
$l_{max} = 0$	□ counter for maximum derivative levels found
$D^l_{pos} = 0$	□ position of first encounter of derivative l on S_{poly}
$D^l_{sub} = 0$	□ number of subdivides before differentiating
$n_{right} = 0$	□ number of right-end real roots
$n_{root} = 0$	□ number of total real roots
$[T_{min}, Tmax] = [0, 1]$	□ domain for all master polynomial segments

1.2 □ ASSIGN the master poly structure M^l_{poly} for $l = 0$
□ for polynomial evaluation and approximation.

$M^0_{poly} = \left\{ N^0 = n, \ \{Y^0_i = c_i, \ \varepsilon_{Y^0_i}\}_{i=0}^{N^0}, \ Y^0_{tol} = \| \ \mathcal{E}_{\|Y^0_i\|}, \ \varepsilon_{Y^0_i} \ \| \right\}$

1.3 □ NORMALIZE $M^{l=0}_{poly}$.

NormPoly (M^l_{poly})

1.4 □ TEST for $M^{l=0}_{poly}$ root(s) at right interval limit $t = T_{max}$.

Bend0$_{right}$ (M^l_{poly}, n_{right})

1.5 □ TEST for $M^{l=0}_{poly}$ root(s) at left interval limit $t = T_{min}$, and
□ RECORD any found.

Bend0$_{left}$ (M^l_{poly}, n_{roots})
RecordRoots (n_{roots} @ $T_{min} \ \rightarrow \ \{r_j\}$)

1.6 □ ASSIGN the local poly structure $L^{l=0}_{poly}$, and
□ PROCEED to 2.1.

$L^0_{poly} \ = \ \left\{ [t^0_0, t^0_n] = [T_{min}, T_{max}], \ \ n^l = N^0, \ \ \{y^l_i = Y^0_i\}_{i=0}^{n^l} \right\}$

Bcom.2: Real Root Isolation Loop Steps

2.1 □ TEST for candidate root(s) at left interval limit $t = t^l_0$.

If ($l > 0$ And $|y^l_0| \leq Y^l_{tol}$)

2.1.1 □ EVALUATE M^l_{poly} at t^l_0.

Beval (t^l_0, M^l_{poly})

2.1.2 □ CONFIRM candidate root t^l_0 when $|Y^l(t^l_0)| \ \leq \ \mathcal{E}_{Y^l(t^l_0)}$.

If ($|Y^l(t^l_0)| \ \leq \ \mathcal{E}_{Y^l(t^l_0)}$)
Bdeflate$_{left}$ ($\{y^l_i\}$)
Proceed to 2.1
EndIf

2.2 □ COMPUTE the number of sign variations n_{cross} in $\left\{y_i^l\right\}_{i=0}^{n^l}$, and
□ TRACK the first two i position crossings as i_{cross_1} and i_{cross_2}.

2.3 □ TEST if control polyline crosses t-axis.

IF ($n_{cross} > 1$) □ subdivide or differentiate L_{poly}^l

2.3.1 □ COMPUTE the local subdivision point τ of poly segment $\{y_i\}_{i=0}^{n^l}$.

$\tau = \frac{(i_{cross_1}+i_{cross_2})+1.5}{2n^l} \in [0,1]$

2.3.2 □ EVALUATE M_{poly}^l at its subdivide point t_{sub},
□ COMPUTING its bounding error $\mathcal{E}_{Y^l(t_{sub})}$.

$t_{sub} = t_0^l + \tau(t_n^l - t_0^l)$
BEVAL (t_{sub}, M_{poly}^l)

2.3.3 □ TEST whether to differentiate L_{poly}^l,

IF ($D_{sub}^l > D_{MAXSUB}$ OR $|Y^l(t)| \leq \mathcal{E}_{Y^l(t)}$)
$D_{pos}^l = S_{top}$
$D_{sub}^l = D_{sub}^{l+1} = 0$
INCREMENT : l
□ COMPUTE a new master derivative M_{poly}^l from M_{poly}^{l-1}.
IF ($l > l_{max}$)
BDERIV ($\{Y_i^{l_{max}}, \varepsilon_{Y_i^{l_{max}}}\}_0^{N^{l_{max}}}$, $\{Y_i^l, \mathcal{E}_{Y_i^l}\}_0^{N^l}$)
INCREMENT : l_{max}
ENDIF
BDERIV_{psedo} ($\{y_i^{l-1}\}_0^{n^{l-1}}$, $\{y_i^l\}^{n^l}$)

2.3.4 □ OTHERWISE SUBDIVIDE L_{poly}^l
□ REPLACING L_{poly}^l with the left poly segment, and
□ PUSHING the right poly segment onto stack S_{poly}.

ELSE
INCREMENT : S_{top} and D_{sub}^l
BSUBDIV_{left} (L_{poly}^l, $S_{poly}^{S_{top}}$)
$\text{ENDIF}_{D_{sub}^l}$

2.4 □ OTHERWISE only 1 or 0 real roots.

ELSE

2.4.1 □ TEST for 1 real root.

IF ($n_{cross} \equiv 1$)
$l_{sub}^l = 0$
□ SET root bounds.

$[t_{left}, t_{right}] = [t_0^l, t_n^l]$
□ APPROXIMATE root to its multiplicity.
PROCEED TO 3.1
ENDIF

2.4.2 □ TEST for termination,
IF ($S_{top} \equiv 0$)
IF ($l = 0$) PROCEED TO 2.5
ELSE DECREMENT : l

2.4.3 □ OTHERWISE POP the top of stack $\{S_{poly}^i\}_{i=1}^{S_{top}}$ to L_{poly}^l.
ELSE
$L_{poly}^l = S_{poly}^{s_{top}}$
DECREMENT : S_{top}
$\text{ENDIF}_{S_{top}}$

$\text{ENDIF}_{n_{cross}}$

2.5 □ RECORD n_{right} roots at T_{max}, and
□ RETURN.
RECORDROOTS (n_{right} @ $T_{max} \rightarrow \{r_j\}$)
EXIT BCOM

BCOM.3: (MULTIPLE) REAL ROOT APPROXIMATION LOOP STEPS

3.1 □ REFINE $t_{root} \in [t_{left}, t_{right}]$ with multiplicity $m = 1$.
$m = 1$
BREFINE ($[t_{left}, t_{right}]$, t_{tol}, $\{Y_i^l, \varepsilon_{Y_i^l}\}_i^{N^l}$, t_{root})

3.2 □ TEST whether to record m roots at t_{root}^l.
IF ($l = 0$)
RECORDROOTS (m @ $t_{root}^l \rightarrow \{r_j\}$)
IF ($t_{right} < t_n^l$)
$\tau = \frac{t_{right} - y_0^l}{y_n^l - y_0^l}$
BSUBDIV_{right} (τ, L_{poly}^l)
WHILE ($|y_0^l| \leq Y_{tol}^l$) BDEFLATE_{left} (L_{poly}^l)
PROCEED TO 2.1
ENDIF

3.3 □ OTHERWISE TEST for possible multiple roots by
□ EVALUATING M_{poly}^l at t_{root}^l, and
□ COMPARING $|Y^l(t_{root}^l)| \leq \mathcal{E}_{y^l(t_{root}^l)}$.

ELSE
DECREMENT : l
$S_{top} = D^l_{pos}$
BEVAL (t^l_{root}, $\{Y^l_i, \varepsilon_{Y^l_i}\}_{i=0}^{N^l}$, $Y(t^l_{root})$, $\mathcal{E}_{Y^l(t^l_{root})}$)

3.3.1 □ TEST for multiple roots at t^l_{root}.
IF ($Y^l(t^l_{root}) \leq \mathcal{E}_{Y^l(t^l_{root})}$)
INCREMENT : m
PROCEED TO 3.2

3.3.2 □ OTHERWISE TEST for root in either the left or right segment related to t^l_{root}.
ELSE
$m = 0$
IF (y^l_0 $Y^l(t^l_{root}) < 0$)
$t_{left} = t^l_0$
$t_{right} = t^l_{root}$
PROCEED TO 3.1
ELSE
INCREMENT : S_{top} $D^l_{sub} = 0$
$\tau = (t^l_{root} - t^l_0) \,/\, (t^l_n - t^l_0)$
BSUBDIV$_{left}$ (τ, $\{y^l\}_{i=0}^{n^l}$, $S_{poly}^{S_{top}}$)
ENDIF
PROCEED TO 2.1
ENDIF$_{Y^l(t^l_{root})}$

ENDIF$_l$

The root isolation heuristic for BCOM is illustrated in Figures 6.3 and 6.4. Figure 6.3.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 6.3.b shows the polynomial after deflating the roots at 0 and 1. In Figure 6.3.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 6.3.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 6.4.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 6.4.f performs another subdivision, with the same results,

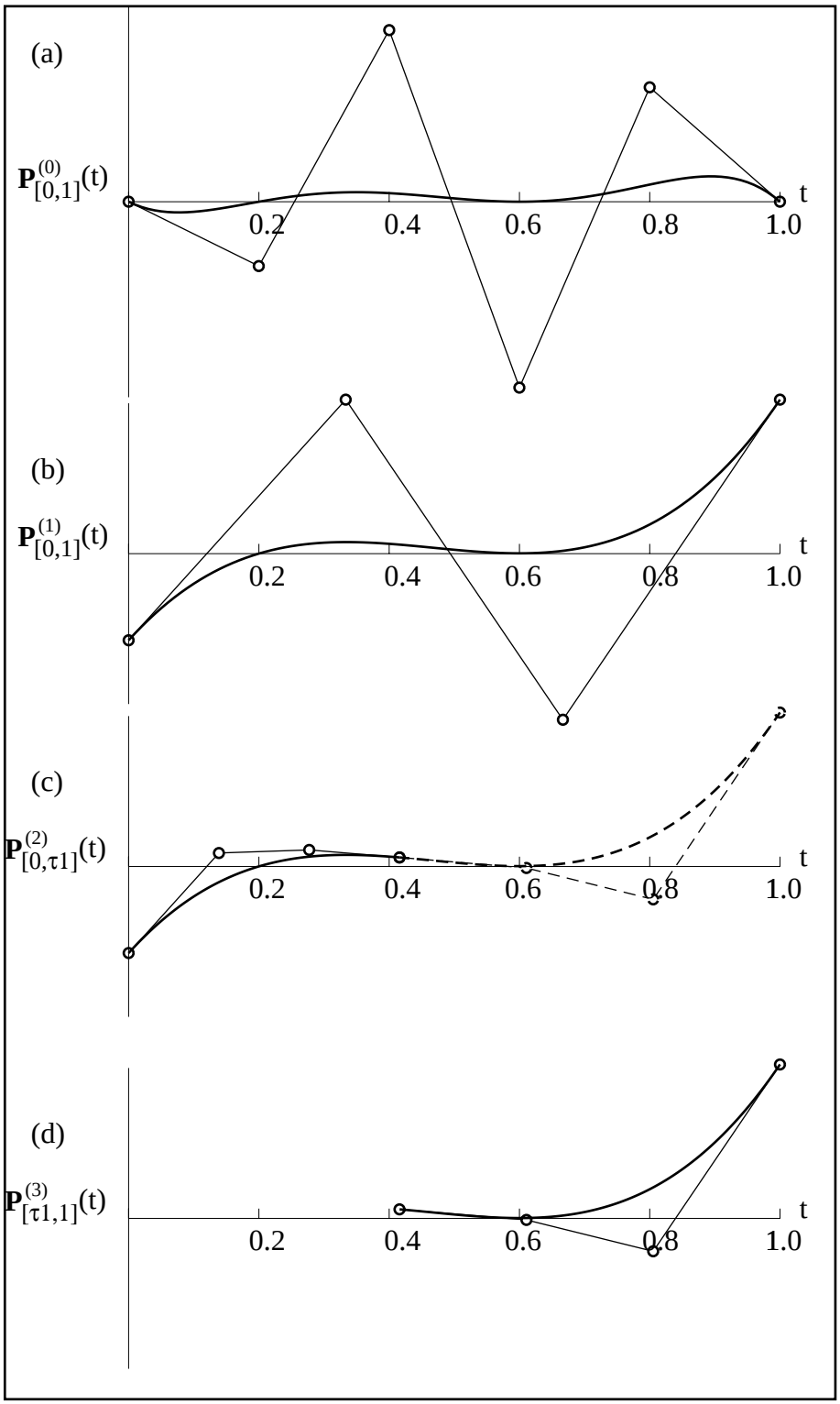

Figure 6.3: BCOM root isolation heuristic (a-d).

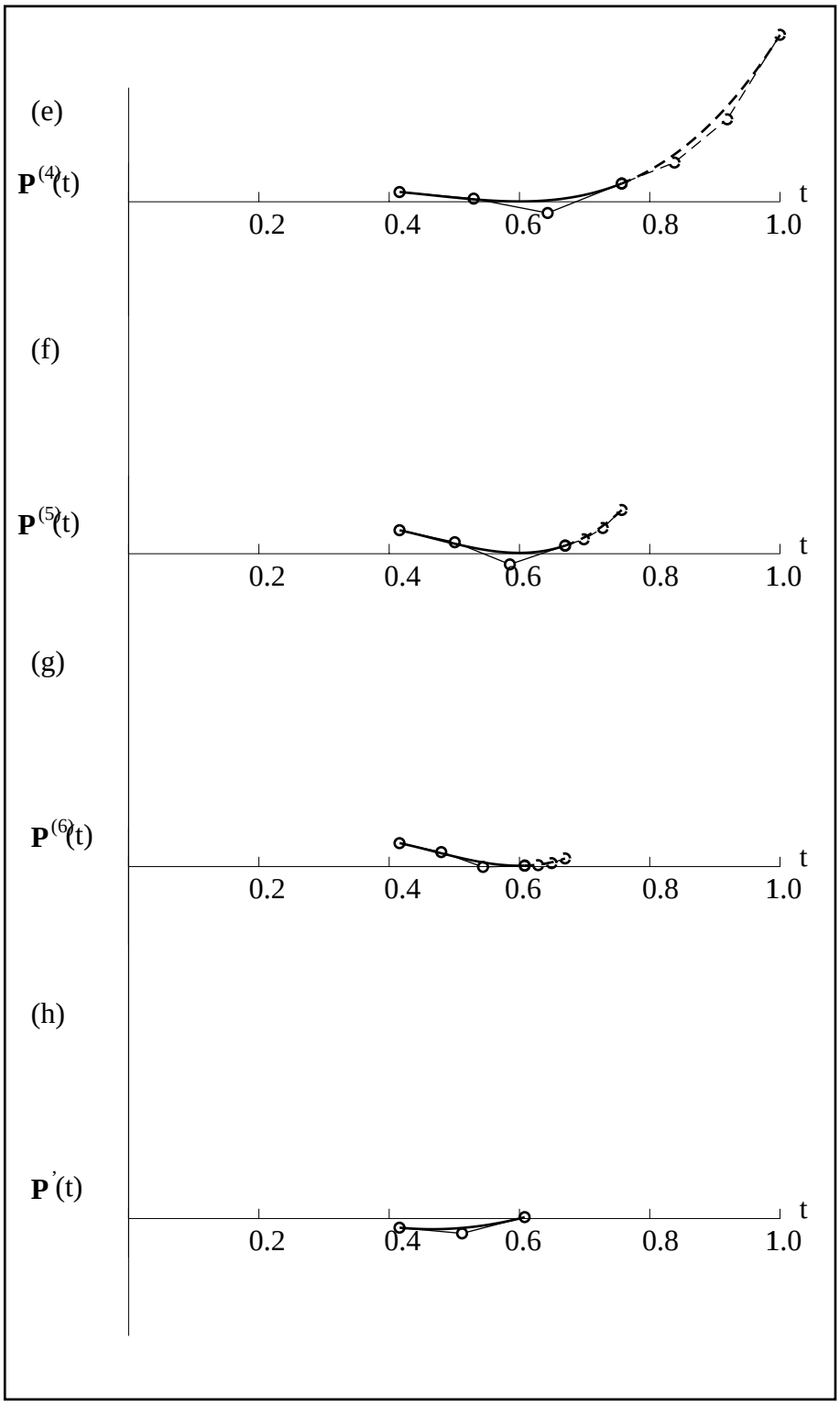

Figure 6.4: BCOM root isolation heuristic (e-h).

as does Figure 6.4.g. At this point, BCOM 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 6.4.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.

6.3.3 BCOM Considerations

BCOM uses three different stacks to store the following polynomial segment structures:

master The master stack contains the original polynomial along with any of its stacked (true) derivative polynomials which are required and only used for any subsequent polynomial evaluation during the isolation (STAGE 2) and approximation (STAGE 3) of its corresponding roots. All of these polynomials denote master polynomials and maintain a unit domain. The initial level master polynomial contains the normalized [§ 5.4] original polynomial after any left or right domain limit roots are detected and then deflated [§ 5.5] during pre-isolation STAGE 1. The master derivatives are computed and stored as deemed necessary during isolation STEP 2.3.3.

A master polynomial structure consists of the degree of the derivative, the coefficients and their errors which accumulate according to linearized running error analysis for each derivative operation [§ 5.6], and a global coefficient tolerance which is the maximum of either the maximum coefficient errors, or the de Casteljau global bound [§§ 5.1].

local The local stack contains a stack of local polynomial segment structures which are appropriately subdivided and differentiated during the real root isolation stage. The local stack is replenished by polynomials on the pending stack, which are appropriately popped and assigned in STEP 2.3.4.

A local polynomial structure consists of the degree of the polynomial segment, the coefficients, and the domain limits over which the polynomial segemnt is defined which are always a subset of the master domain. The local coefficients are each initially scaled by their corresponding binomial factor to minimize calculations in the evaluation procedures.

pending The pending stack contains any a series of "remaining" subdivided right local polynomial segments that need to be processed.

The local and subdivide stacks are used extensively during root isolation and multiple root detection loops.

Note the conditions in STEP 2.3.3 which qualify the polynomial for differentiation as opposed to subdivision. The basic strategy employed is when in doubt to whether the convergence is slowing or an isolation step approximates a root, simply differentiate. Setting the constant $D_{MAXSUB} = 2$ is inexpensive and provides good results, as opposed to larger numbers which usually do not elevate the isolation step into the proper derivative levels for accurate and efficient multiple root approximation. BCOM exhibits efficient and accurate convergence once it determines which derivative contains the linear factor of the multiple root. These heuristics also force all root approximations to be performed in stage three which minimizes special case root tests.

In addition, polynomial residual error tests are employed in STEPS 1.4, 1.5, 2.1.2, 2.3.3, 3.3, and 3.3.1, as opposed to constant error tests which are usually too conservative for multiple or clustered root considerations. Although global error tolerances do provide efficient candidate checks for possible use of computed running error bounds as demonstrated in STEP 2.1.

Deflation is employed to factor out the leading zeros of only derivative polynomials, otherwise it is avoided. This helps maintain unperturbed master coefficients for polynomial evaluation purposes (see STEPS 2.1 and 3.2).

The isolation step heuristic in STEP 2.3.1 is illustrated in Figure 6.5 where the boxed coefficients, $\mathbf{P}_0$ and $\mathbf{P}_3$, represent the pair of variables i_{cross_1} and i_{cross_2}, respectively, and the $\times$ at 0.25 represents the point of subdivision.

An algorithm to approximate and purify isolated real roots is represented by

$$\text{BREFINE}\ (\ [t_{left}, t_{right}],\ t_{tol},\ \{Y_i^l, \varepsilon_{Y_i^l}\}_i^{N^l},\ t_{root}\ )$$

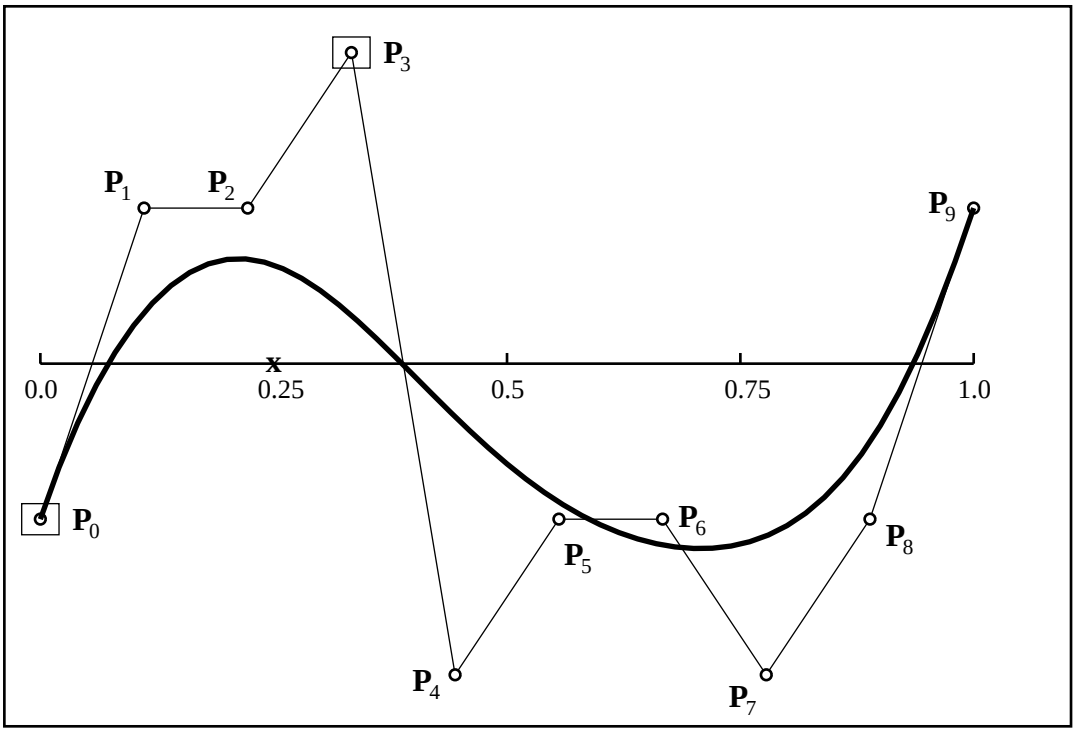

Figure 6.5: BCOMsubdivision step heuristic.

where modified regula falsi first computes an initial approximate root using the scaled local coefficients. The initial root is then purified a second time by modified regula falsi using the scaled master polynomial coefficients.

Finally, the following function is considered for recording roots.

Algorithm 6.3.3.1 (RECORDROOTS (n @ $x \rightarrow \{r_j\}_{j=1}^{n_{roots}}$)) *Record and return n roots at value x into array $\{r_j\}$ and increment the current root counter n_{roots} by n, whereupon the calling function is EXITED ONLY when the leftmost root only flag $LeftRoot_{only}$ is defined and true in the calling function.*

1. $\{ r_j = value \}_{j=n_{roots}+1}^{n_{roots}+n}$
2. $n_{roots} = n_{roots} + n$
3. IF ($LeftRoot_{only}$) EXIT *the calling function*

END OF RECORDROOTS

The left–most–root flag is useful for some applications such as ray tracing in which only the smallest positive root is of interest.

6.4 Bernstein Convex Hull Isolating Step (BCHI)

The *Bernstein convex hull isolating step* (BCHI) algorithm separates each real root of $[0, 1]$ in ascending order by stepping along the t-axis in increments which are guaranteed to contain at most one root. The step size is determined by taking the leftmost edge of an optimal convex hull formed from a linear combination of the Bernstein polynomial segment and its derivative. Once isolated, each root of each subinterval is subsequently refined the modified regula falsi method. The following subsections describe the preliminary concepts and outline the basic strategy of algorithm BCHI.

6.4.1 Preliminary Concepts

Defining the original polynomial as

$$\mathbf{P}_{[a,b]}(t) = (t, y_{[a,b]}(t)) \tag{6.4.1.5}$$

and its first derivative polynomial as

$$\mathbf{P}'_{[a,b]}(t) = (t, y'_{[a,b]}(t)) \tag{6.4.1.6}$$

we have the following theorem.

Theorem 6.4. Assume that $\mathbf{P}_{[a,b]}(t)$ has at least two roots in $[a,b]$. Denote by τ_1 the leftmost root, and by τ_5 the next largest root (τ_2, τ_3, and τ_4 will be used in the proof). Form a new polynomial

$$\mathbf{V}_{[a,b]}(t) = (t, v_{[a,b]}(t)) = (t, c_1 y_{[a,b]}(t) + c_2 y'_{[a,b]}(t)) \tag{6.4.1.7}$$

where c_1 and c_2 are any real numbers, not both zero. For any give c_1 and c_2, denote the leftmost root of $\mathbf{V}_{[a,b]}(t)$ by $V_L(c_1, c_2)$. For all possible c_1 and c_2, denote the largest value of $V_L(c_1, c_2)$ by $\max(V_L)$ Then,

$$\tau_1 \leq \max(V_L) \leq \tau_5. \tag{6.4.1.8}$$

Proof: . τ_1 is a lower bound for $\max(V_L)$ because $V_L(1,0) = \tau_1$. If $\tau_1 = \tau_5$ (a multiple root), then $\mathbf{P}_{[a,b]}(t)$ and $\mathbf{P}'_{[a,b]}(t)$ both have τ_1 as their leftmost root, and $\max(V_L) = \tau_1$.

If $\tau_1 \neq \tau_5$, then by Rolle's theorem, $\mathbf{P}'_{[a,b]}(t)$ has at least one root $\tau_3 \in [\tau_1, \tau_5]$. Assume without loss of generality that $y(a) < 0$. Then, since $\mathbf{P}_{[a,b]}(t)$ and $\mathbf{P}'_{[a,b]}(t)$ are continuous functions, there exists a value $\tau_2 \in [\tau_1, \tau_3]$ such that $y(\tau_2) = y'(\tau_2)$ and their also exists a value $\tau_4 \in [\tau_3, \tau_5]$ such that $y(\tau_4) = -y'(\tau_4)$.

Again since $\mathbf{P}_{[a,b]}(t)$ and $\mathbf{P}'_{[a,b]}(t)$ are continuous functions,

1. there is at least one root for $v(t)$ in $[\tau_1, \tau_2]$ for $c_1 = 1$ and $c_2 \in [-1, 0]$.

2. there is at least one root for $v(t)$ in $[\tau_2, \tau_3]$ for $c_2 = 1$ and $c_1 \in [-1, 0]$.

3. there is at least one root for $v(t)$ in $[\tau_3, \tau_4]$ for $c_2 = 1$ and $c_2 \in [0, 1]$.

4. there is at least one root for $v(t)$ in $[\tau_4, \tau_5]$ for $c_1 = 1$ and $c_2 \in [0, 1]$.

Therefore, $v(t)$ has at least one root in $[\tau_1, \tau_5]$ for any c_1 and c_2, and $V_L \leq \tau_5$, with equality holding only if $\tau_1 = \tau_5$. ■

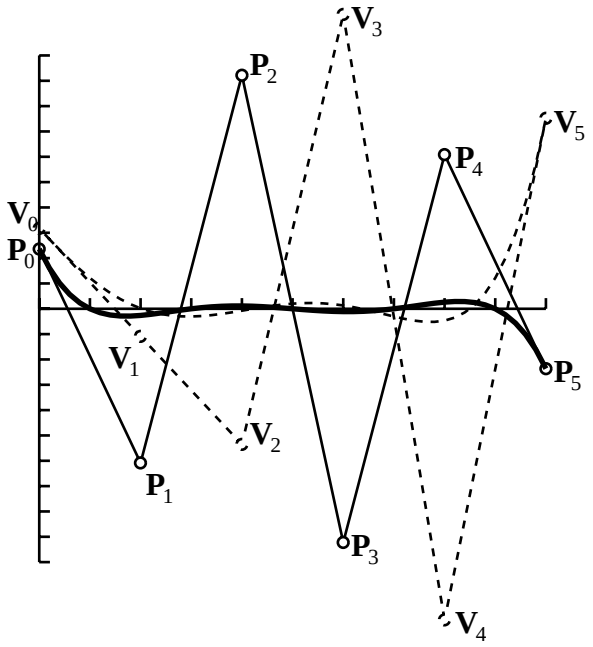

Figure 6.6: Root isolation using BCHI

We can use this theorem to devise an algorithm for determining a large step size which will not pass through more than one root. In essence, the algorithm considers all possible values of c_1 and c_2 and finds the maximum value of where the convex hulls of all the resulting $\mathbf{V}_{[a,b]}(t)$ intersect the t-axis. This is an optimization problem which reduces to linear programming by observing that the optimal choice of c_1 and c_2 is one for which the leftmost edge of the convex hull of the control points of $\mathbf{V}_{[a,b]}(t)$ contains three collinear points. This search can be performed in linear time.

If we are lucky, the convex hull of the control points of $\mathbf{V}_{[a,b]}(t)$ will intersect the t axis between τ_1 and τ_5, thereby isolating the leftmost root of $\mathbf{P}_{[a,b]}(t)$. An example of this is given in Figure 6.6. Notice in this example that $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$ are collinear.

6.4.2 Algorithm: BCHI

BCHI is similar to the BCHA algorithms except it uses an *isolating* as opposed to an *approximating* convergence step to bound a real root. The following algorithm discusses the steps of BCHI which are outlined below.

Algorithm 6.4.2.1 (BCHI ($LeftRoot_{only}$, t_{tol}, $\{c_i, \varepsilon_{c_i}\}_{i=0}^{n}$, $\{r_j\}_{j=1}^{n_{roots}}$)) BCHI *receives as input a leftmost–root–only flag* $LeftRoot_{only}$, *a solution tolerance* t_{tol}, *and Bernstein coefficients and their errors* $\{y_i, \varepsilon_{y_i}\}$ *defined over the domain* $[0, 1]$ *and returns* n_{roots} *roots* $\{r_j\}$.

BCHI.*1.* REAL ROOT PRE-ISOLATION STEPS,

BCHI.*2.* REAL ROOT ISOLATION STEPS.

END OF BCHI

BCHI.1: REAL ROOT PRE-ISOLATION STEPS

1.1 INITIALIZE counters,
ASSIGN the interval tolerance t_{tol},
ASSIGN a master polynomial (poly) structure.
(The master poly structure consists of parameter limits $[T_{min} = 0., T_{max} = 1.]$, degree N, coefficients Y_i, and error coefficients $E_i = 0.$; and is used for any root refinement evaluation.)

1.2 NORMALIZE the coefficients Y_i.

1.3 TEST for rightmost root(s) at the right interval limit T_{max}.
(IF ($|Y_n| \leq \mathcal{E}_{Y_n}$), a root exists at T_{max}. Thereupon, INCREMENT the rightmost root counter n_{right}, DEFLATE the master poly at T_{max}, and REPEAT this step, testing for further (multiple) roots at T_{max}. Use: BEND0_{right})

1.4 COPY master poly structure to local poly structure,
PROCEED to STEP 2.1.
(The local poly structure consists of parameter limits $[t_0, t_n]$, degree n, coefficients y_i, and error coefficients e_i.)

BCHI.2: REAL ROOT ISOLATION STEPS

2.1 TEST for root(s) at the left interval limit t_0.
(IF ($(|y_0| \leq e_0)$ OR $delta_{flag}$), a root exists at $t_{root} = t_0$. Therefore, RECORD t_{root}, DEFLATE the local poly at t_{root}, and REPEAT this step, checking for possible (multiple) roots at t_{root} before continuing.)

2.2 COMPUTE the number of sign variations n_{vary} in the y_i sequence.
(The value n_{vary} determines the number of possible real roots. It is implied that IF ($n_{vary} = 0$), the local poly has no real roots; that is, the master poly has no real roots in the interval $[t_0, t_n]$, and thus PROCEED to STEP 2.5.)

2.3 TEST for > 1 real roots.
(IF ($n_{vary} > 1$), need to find an optimal step to subdivide the local poly so that only its leftmost root is isolated.)

2.3.1 COMPUTE a conservative approximation $t_{step} = max(\mathbf{V}_L(c_1, c_2))$, TEST for convergence of the step.
(IF ($t_{step} * (t_n - t_0) \leq t_{tol}$), ASSIGN $delta_{flag} = TRUE$, and PROCEED to STEP 2.1. In addition, the computation of t_{step} also provides a *possible* slow convergence flag *split* for STEP 2.3.4.)

2.3.2 SUBDIVIDE the local poly at t_{step}.
(ASSIGN the left poly segment to a left poly structure with limits $[t_l = t_0,\ t_r = t_0 + t_{step}(t_n - t_0)]$, coefficients yl_i, and error coefficients el_i; and the right poly segment as the local poly with the parameter limits $[t_0 = t_r,\ t_n]$. The left poly structure is used for the next step's REFINE.)

2.3.3 TEST for root in left segment.
(Essentially, IF ($yl_0 * yl_n < 0$), the left segment contains an isolated real root. Therefore, REFINE the real root t_{root} within the limits $[t_l, t_r]$, RECORD t_{root}, DEFLATE the local poly at t_{root}, and PROCEED to STEP 2.1.)

2.3.4 TEST for possible slow convergence.
(IF (*split*), there is a *possible* several orders of magnitude variance among the y_i. Therefore, SUBDIVIDE$_{left}$ the local poly at t_{split} where $[t_0 = t_{split},\ t_n = t_0 + t_{split}(t_n - t_0)]$.)

2.3.5 PROCEED to STEP 2.1.

2.4 TEST for 1 real root.
(ELSE IF ($n_{vary} = 1$), REFINE the isolated real root t_{root} between the limits $[t_0, t_n]$, and RECORD it.)

2.5 TEST for an inner poly segment.
(IF ($t_n < T_{max}$), the local poly is an inner poly segment. Therefore, ASSIGN $[t_0 = t_n,\ t_n = T_{max}]$, $n = N$, and $y_i = Y_i$. SUBDIVIDE$_{right}$ the local poly at t_0, and PROCEED to STEP 2.1.)

2.6 RECORD n_{right} roots = T_{max}, and RETURN.

BCHIalso uses modified regula falsi to REFINE the bracketed real roots. RECORDROOTS is described in [§6.3.2].

6.5 Standard Isolator Polynomial Separation (SIPS)

The *standard isolator polynomial separation algorithm* (SIPS) solves power form polynomials of degrees 1–9. This algorithm is an extension of a degree 6 scheme originally developed by Klimazewski and Sederberg which is based on Sederberg-Chang's [SC94] concepts regarding *isolating polynomials* [§§ 2.2.2] called *isolator polynomials*, or IPs, which are revisited below.

SIPS finds the real roots of a polynomial in ascending order by appropriately sorting lower and upper real root bounds along with the roots of the generated IPs, and using these values as the corresponding interval limits for each distinct real root of the original polynomial. The roots of the IPs are solved using closed form real root solvers [§§ 2.3.1,§ 5.9]. The coefficients of the IPs are formed from closed form equations that are derived from a pair of IPs which are generated from a polynomial remainder sequence of the polynomial and its first derivative.

An effort was attempted to formulate IPs in Bernstein form, hoping they would reduce to a more intuitive, eloquent, and efficient form. Since achieving this desired objective proved unsuccessful, the algorithm is presented for polynomials in power form as to aide in future discussion regarding testing with implementation of power form root finders presented in Chapter 7 on numerical results.

6.5.1 Preliminary Concepts

The SIP algorithm exploits the fact that the real roots of the two IPs $a(x)$ and $b(x)$ isolate all the real roots of the given standard polynomial $p(x)$. Thus, all the real roots of a given standard polynomial $p(x)$ are isolated by solving for the roots of the two IPs $a(x)$ and $b(x)$ which are generated from a polynomial remainder sequence generated by $p(x)$ and $p'(x)$. Since the roots of the IPs are obtained via standard closed-form real root solvers, this algorithm can find the real roots of a given polynomial up to degree 9 where both IPs would be degree 4. These isolating roots are then sent as interval limits to a bracketing root approximation function.

Isolator Polynomials

This section contains a brief description of Sederberg and Chang's report on isolating polynomials which separate the real roots of a polynomial p(x) in $\Re[x]$ by locating intervals that contain exactly one real root of p. The method consists of finding a pair of auxiliary polynomials whose set of combined real roots contain at least one value in every closed interval defined by each pair of adjacent real roots in p. It turns out that any member of the polynomial remainder sequence generated by p and p' can serve as one of these auxiliary polynomials. The balance of the following discussion is an abridgement of Sederberg and Chang's report [SC94]. Refer to this report for motivating examples.

Isolator polynomials are a pair of auxiliary polynomial defined as $a(x)$ and $b(x)$ in the following theorem [SC94].

Theorem 6.5.1.1 (Isolator Polynomials, IP) *Given any polynomial $p(x) \in \Re[x]$ with two adjacent distinct real roots α_1 and α_2, and given any two other polynomials $b(x), c(x) \in \Re[x]$, define*

$$a(x) = b(x)p'(x) + c(x)p(x), \qquad (Deg(p) = n, a(x) \in \Re[x]) \tag{6.5.1.9}$$

Then $a(x)$ or $b(x)$ has at least one distinct real root in the closed interval $[\alpha_1, \alpha_2]$.

Thus, between any pair of distinct real roots of $p(x)$ lies a distinct real root of $a(x)$ or $b(x)$.

It is always possible to generate a pair of IPs $a(x)$ and $b(x)$ where

$$Deg(a) + Deg(b) \;=\; Deg(p) - 1$$

from a polynomial remainder sequence based on p and p'. For instance, let $r_i(x)$ define any member of the Sturm's sequence [§ 2.2.2] of p and p', that is

$$\begin{array}{lcll} r_{-1}(x) & = & p(x) & \\ r_0(x) & = & p'(x) & \\ r_1(x) & = & r_{-1}(x) - q_1(x) r_0(x) & \\ r_2(x) & = & r_0(x) - q_2(x) r_1(x) & \\ & \cdots & & \\ r_i(x) & = & r_{i-2}(x) - q_i(x) r_{i-1}(x), & (r_0 = p'(x),\ r_{-1} = p(x)). \end{array}$$

Then $a_i(x)$, $b_i(x)$ and $c_i(x)$ are formed in general by

$$\begin{array}{lcll} a_i(x) & = & r_i(x), & (a_0 = r'(x), a_{-1} = r(x), a_{-2} = 0) \\ b_i(x) & = & b_{i-2}(x) - q_i(x) b_{i-1}(x) & (b_0 = 1, b_{-1} = 0), \\ c_i(x) & = & c_{i-2}(x) - q_i(x) c_{i-1}(x), & (c_0 = 0, c_{-1} = 1) \end{array}$$

with $Deg(a_i) \leq Deg(p) - i - 1$ and $Deg(a_i) + Deg(b_i) = Deg(p) - 1$.

The following recursive *Mathematica* [Wol91] expressions [Sed93] establish such a sequence based upon $p(x)$ and $p'(x)$ to generate a pair of IPs $a(x)$ and $b(x)$ and the polynomial $c(x)$, where px represents the polynomial equation of $Deg(p(x)) = n$ in standard form.

Algorithm 6.5.1.1 (*Mathematica* Power Form IPs) *Isolator polynomials represented by $a(x)$ and $b(x)$ are generated from polynomial $px = p(x)$ of degree n in power form.*

```
a[-2,px_] := 0
a[-1,px_] := px
a[0,px_] := D[px,x]
a[n_Integer, px_] := PolynomialRemainder[a[n-2,px],a[n-1,px],x]
q[n_Integer,px_] := PolynomialQuotient[a[n-2,px],a[n-1,px],x]
b[-1,px_] := 0
b[0,px_] := 1
```

```
b[n_Integer,px_] := b[n-2,px] - q[n,px] b[n-1,px]
c[-1,px_] := 1
c[0,px_] := 0
c[n_Integer,px_] := c[n-2,px] - q[n,px] c[n-1,px]
```

END OF *Mathematica* Power Form IPs

The IP coefficients are thus defined completely in terms of the the coefficients of $p(x)$ in closed form.

Theorem (6.5.1.1) addresses the case of distinct real roots. For multiple real roots, $\alpha_1 = \alpha_2$ and $p(\alpha_1) = p'(\alpha_1) = a(\alpha_1)$. Thus, if a root in $a(x)$ coincides with a root in $p(x)$, this root is a multiple root in $p(x)$.

Informal testing performed by [SC94] indicated that IP root isolation is generally less efficient in floating point arithmetic than heuristic methods like Collins and Akritas [CA76] which is based on Descartes' rule of signs.

6.5.2 Algorithm: SIPS

The following three stages are used in conjunction with evaluating the SIPS algorithm. The SIPS algorithm actually constitutes STEP 2, Although any power form polynomial could be substituted for SIPS in STEP 2.

Algorithm 6.5.2.1 (SIPS ($LeftRoot_{only}$, $t_l, t_r, t_{tol}, \{y_i, \varepsilon_{y_i}\}_{i=0}^{n}, \{r_j^{(IP)}\}_{j=1}^{n_{roots}^{(IP)}}$, $\{r_j\}_{j=1}^{n_{roots}}$))
SIPS *receives as input a leftmost–root–only flag* $LeftRoot_{only}$, *solution interval limits* $I[t_l, t_r]$, *a solution tolerance* t_{tol}, *Bernstein coefficients* $\{y_i, \varepsilon_{y_i}\}$, *and returns* $n_{roots}^{(IP)}$ *isolator roots* $\{r_j^{(IP)}\}$ *as well as* n_{roots} *real roots* $\{r_j\}$ *of the original polynomial over the interval* $I[t_l, t_r]$.

SIPS.*1.* REAL ROOT PRE-SIPS STEPS,
(For polynomials input in Bernstein form.)

SIPS.*2.* REAL ROOT ISOLATION STEPS,
(For polynomials in power form.)

SIPS.*3.* REAL ROOT POST-SIPS STEPS.
(For polynomials input in Bernstein form when using pseudo Bernstein conversion.)

END OF SIPS

SIPS.1: REAL ROOT PRE-ALGORITHM STEPS

1.1 ASSIGN a Bernstein polynomial master (poly) structure.
(The master poly structure consists of parameter limits [$T_{min} = 0.$,$T_{max} = 1.$], degree N, coefficients Y_i, and error coefficients E_i.)

1.2 ASSIGN an interval tolerance t_{tol},
COMPUTE a global coefficient tolerance Y_{tol}.

1.3 TEST for rightmost root(s) at the right interval limit T_{max}.
(Use: BEND0$_{right}$)

1.4 TEST for leftmost root(s) at the left interval limit T_{min}.
(Use: BEND0$_{left}$)

1.5 ASSIGN minimum and maximum real root bounds $[t_l, t_r]$.

1.6 CONVERT coefficients from Bernstein Y_i to pseudo power p_i form.

SIPS.2: REAL ROOT ISOLATION STEPS

2.1 NORMALIZE the coefficients p_i.

2.2 COMPUTE the coefficients a_i and b_i of the two isolator polynomials $a(t)$ and $b(t)$.
(The isolator polynomial coefficients a_i and b_i are calculated from coded formulas that are a function of the coefficients p_i from precomputed expressions.

2.3 SOLVE for the real roots of the two isolator polynomials $a(t)$ and $b(t)$ using closed form standard real root solvers.

2.4 SORT the solved isolator roots with the lower and upper real root bounds t_l and t_r.

2.5 REFINE and RECORD the real roots between isolated root's intervals.

2.6 PROCEED TO STEP 3.1.

SIPS.4: REAL ROOT POST-ALGORITHM STEPS

4.1 MAP real roots obtained from pseudo-power coefficients back to the Bernstein interval $[T_{min}, T_{max}]$.

4.2 RETURN.

SIPS uses modified regula for REFINE to approximate the roots, and Horner's algorithm to evaluate a polynomial in power form. Horner's algorithm is implemented both with and without running error analysis which computes bounds for the computed value.

6.5.3 Algorithm Considerations

The following considerations apply to the above SIPS algorithm outline.

Restricting Input to Power Form Polynomials

Polynomials input in power form require the modification of only the following two steps from stage SIPS.1,

SIPS.1.2 ASSIGN an interval tolerance t_{tol},
SIPS.1.5 ASSIGN real root search interval bounds $[t_l, t_r]$.

Steps SIPS.1.1–4 dealing with the Bernstein form of the polynomial and testing for roots at the parameter limits $[T_{min}, T_{max}]$ is not applicable in the power basis since the parameter now takes on the range $t \in [-\infty, +\infty]$. This forces the use of classical real root bounding techniques [§§ 2.2.1] for step SIPS.1.5 when the root search interval bounds $[t_l, t_r]$ are not known.

Although these classical techniques provide analytically sound real root bounds, numerically these values are too conservative and inflict slow convergence. Since these root

bounds $[t_l, t_r]$ consequently bound the leftmost and rightmost real roots, lack of sufficiently close values usually impairs the convergence rate of any serial iterative root refining algorithm. The closeness of these root bounds $[t_l, t_r]$ to their respective leftmost and rightmost real roots is imperative for quick convergence to these roots for any serial iterative root refinement scheme.

Finally, the lack for basis conversion eliminates steps SIPS.1.6 and SIPS.3.1. Therefore, in addition to the two preliminary steps SIPS.1.2 and 5, only stages SIPS.2 and SIPS.3 are required.

Tighter Root Bounds from the Convex Hull

If all the roots in $[T_{min}, T_{max}]$ are sought, then default root search bounds are obviously $[t_l = T_{min}, t_r = T_{max}]$. Tighter bounds result when assigning the left h_l and right h_r convex hull intersections with the t-axis as the root search interval bounds, i.e. $[t_l = h_l, t_r = h_r]$. These hull intersections provide smaller isolation intervals when refining the first and last real roots of the interval.

Pseudo-basis Conversion

Step SIPS.1.6 performs a pseudo Bernstein conversion on the Bernstein coefficients, resulting in pseudo power coefficients [§ 5.8].

Common Roots

Common roots of the original polynomial with the isolator polynomials do not guarantee the existence of multiple roots. Rather, multiple roots are guaranteed to be common with the original polynomial. Thus, it is possible for the isolator roots to coincide with each other and the original polynomial and not be multiple roots. Figure 6.15 depicts a degree 7

Wilkinson polynomial which is a classical example of this artifact. Thus, special techniques are required to separate the coincident isolating interval information which is not covered in this work.

It is important to note that equi-modular root conditions (e.g. the Wilkinson polynomial) tend to produce more cases of common roots among the IPs. Mapping the domain of the polynomial to a different interval (such as in pseudo-basis conversion [§ 5.8]) tends to alleviates detected problems, although "a priori" knowledge of the root distribution is usually not known.

6.5.4 Illustrative Examples of IPs

Below are some illustrative examples of isolator polynomials for various degrees based on given polynomials.

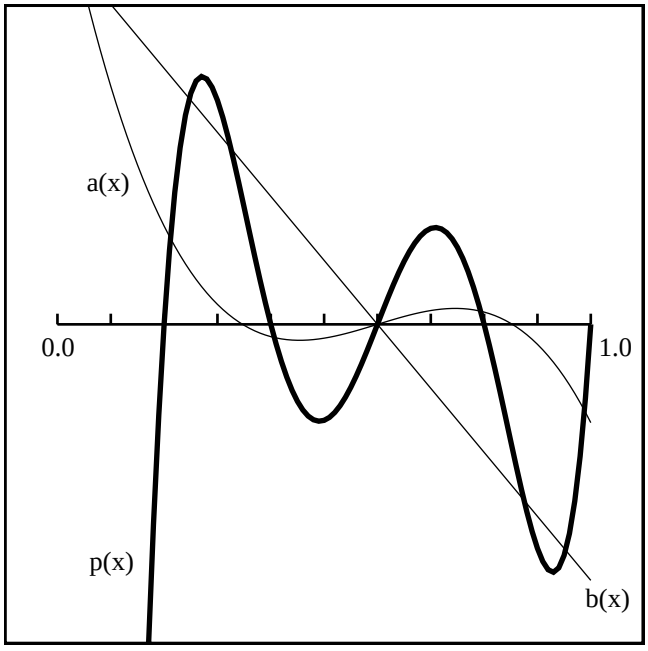

Figure 6.7: IPs of degree 1 and 3 based on a degree 5 Wilkinson polynomial over $I[0,1]$.

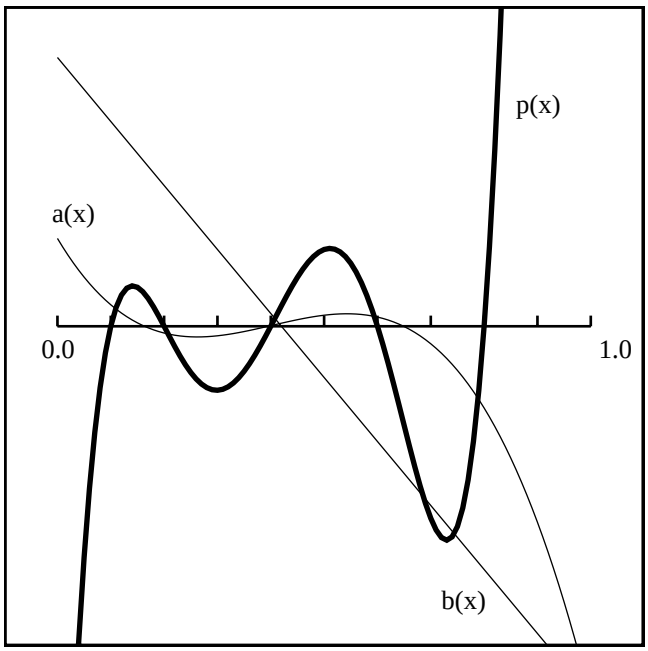

Figure 6.8: IPs of degree 1 and 3 based on a degree 5 polynomial over $I[0,1]$ with roots at $\{.1, .2, .4, .6, .8\}$.

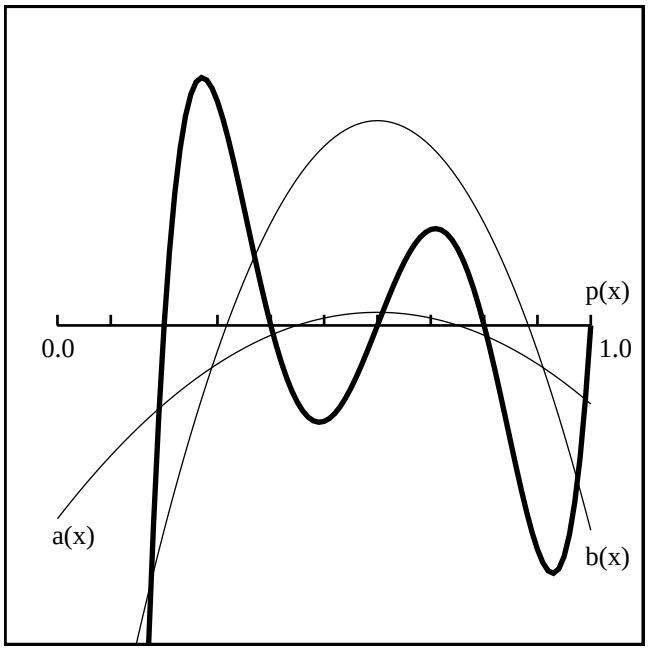

Figure 6.9: IPs both of degree 2 based on a degree 5 Wilkinson polynomial over $I[0,1]$.

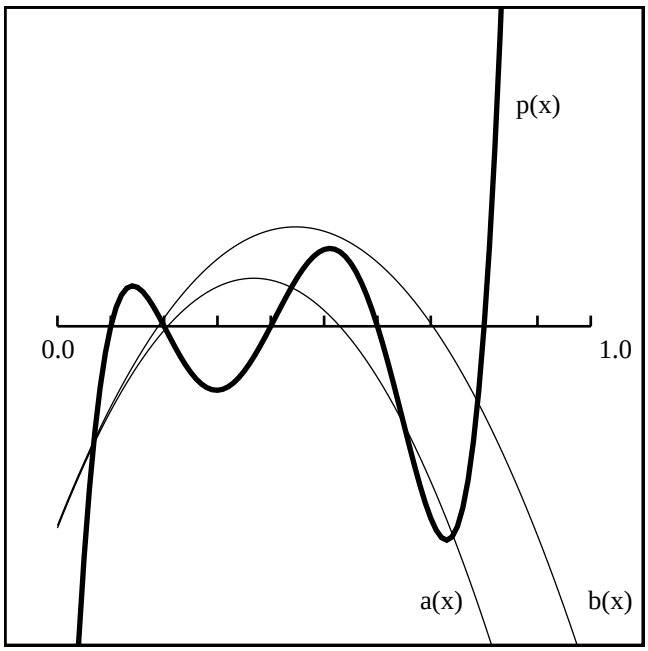

Figure 6.10: IPs both of degree 2 based on a degree 5 polynomial over $I[0,1]$ with roots at $\{.1,.2,.4,.6,.8\}$.

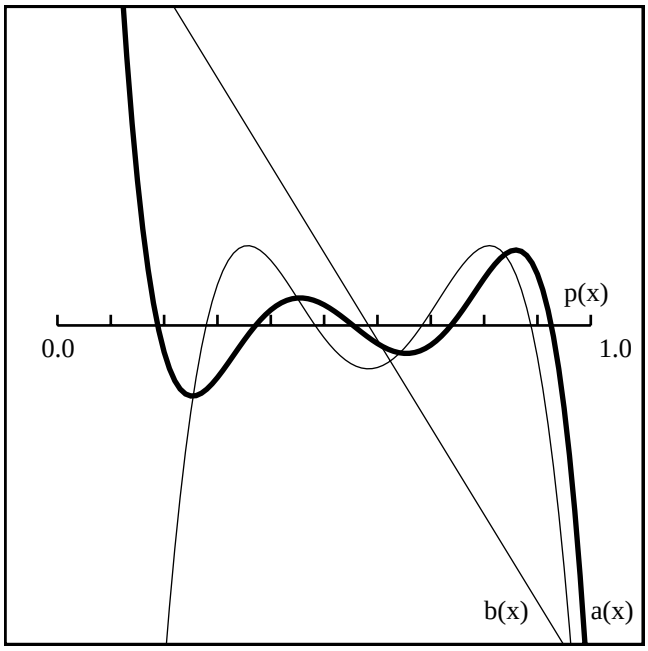

Figure 6.11: IPs of degree 1 and 4 based on a degree 6 Wilkinson polynomial over $I[0,1]$.

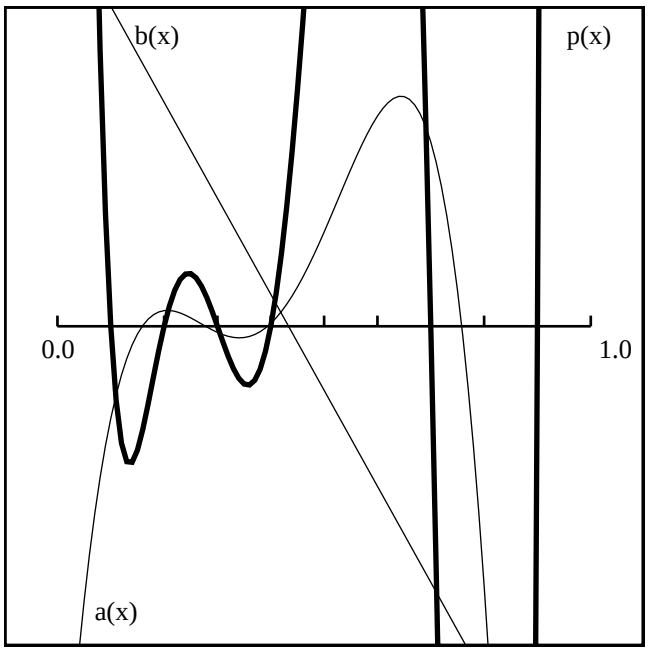

Figure 6.12: IPs of degree 1 and 4 based on a degree 6 polynomial over $I[0,1]$ with roots at $\{.1,.2,.3,.4,.7,.9\}$.

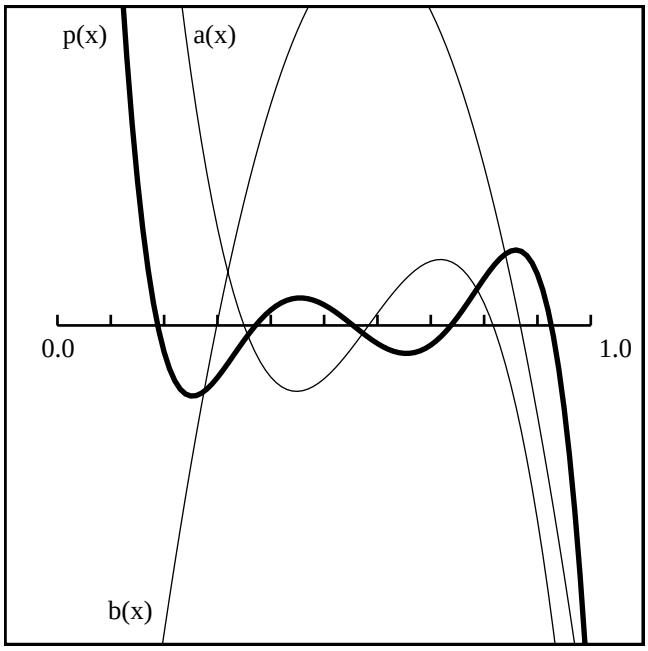

Figure 6.13: IPs of degree 2 and 3 based on a degree 6 Wilkinson polynomial over $I[0,1]$.

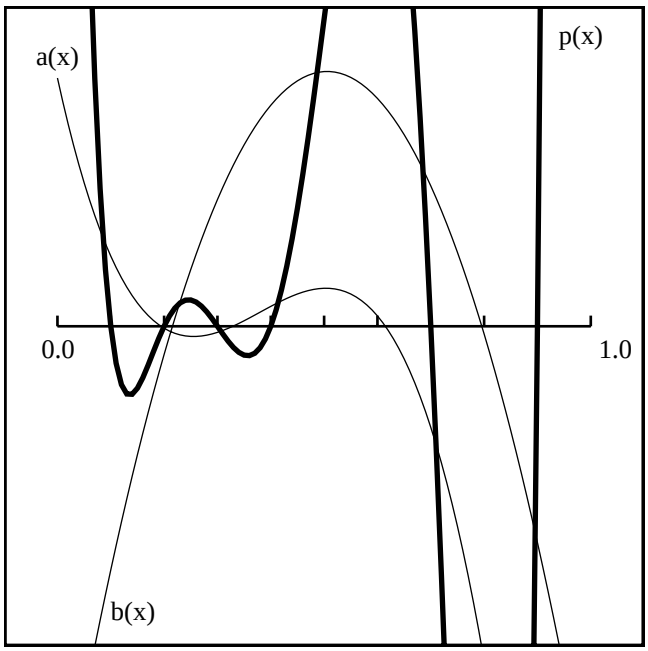

Figure 6.14: IPs of degree 2 and 3 based on a degree 6 polynomial over $I[0,1]$ with roots at $\{.1,.2,.3,.4,.7,.9\}$.

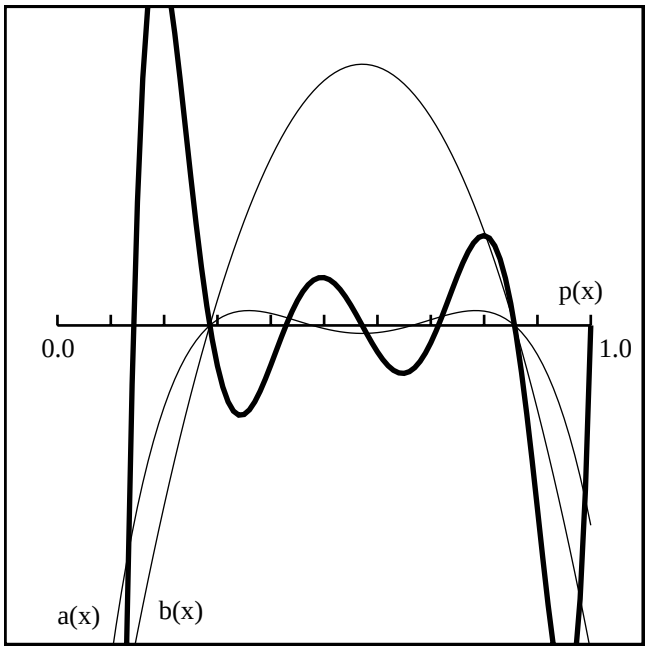

Figure 6.15: IPs of degree 2 and 4 based on a degree 7 Wilkinson polynomial over $I[0,1]$.

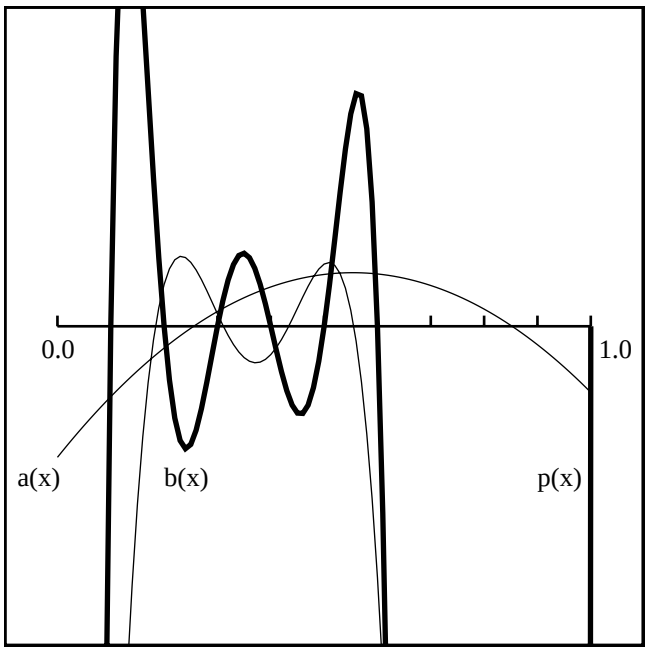

Figure 6.16: IPs of degree 2 and 4 based on a degree 7 polynomial over $I[0,1]$ with roots at $\{.1,.2,.3,.4,.5,.6,1.\}$.

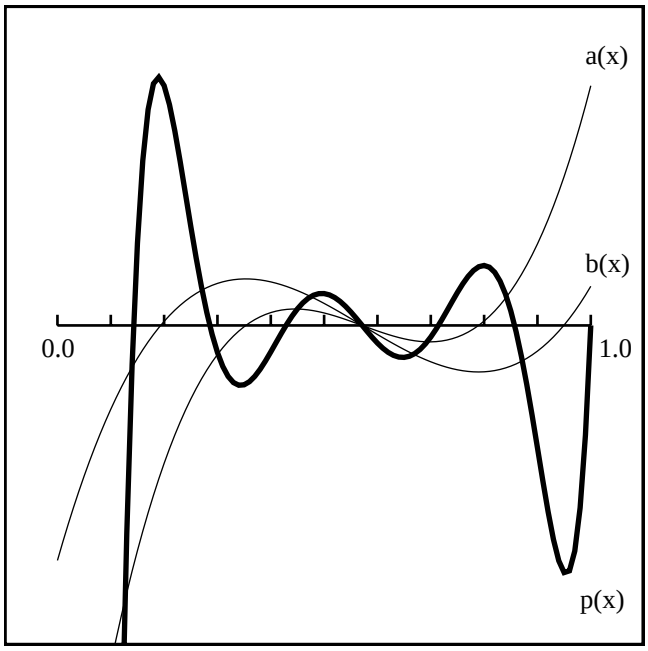

Figure 6.17: IPs both of degree 3 based on a degree 7 Wilkinson polynomial over $I[0,1]$.

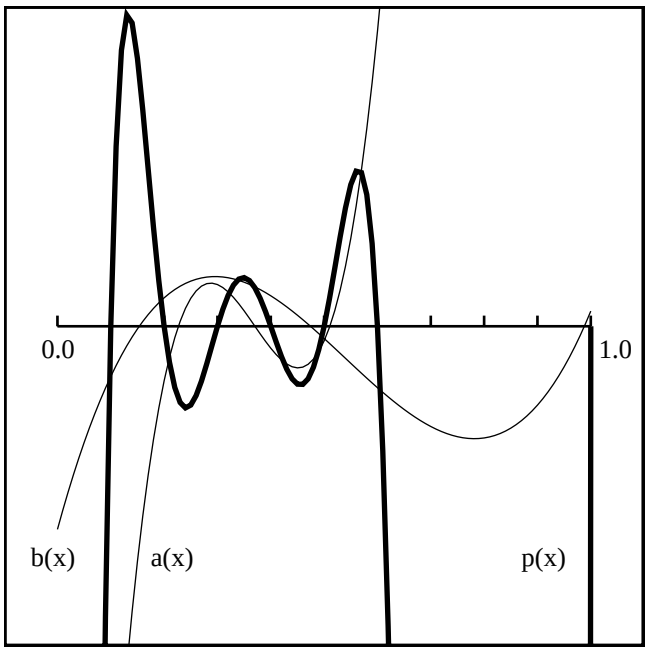

Figure 6.18: IPs both of degree 3 based on a degree 7 polynomial over $I[0,1]$ with roots at $\{.1,.2,.3,.4,.5,.6,1.\}$.

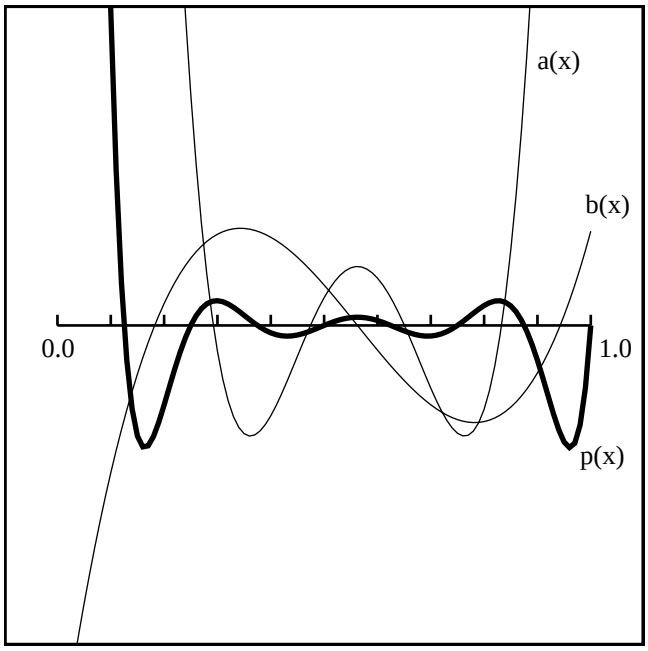

Figure 6.19: IPs of degree 3 and 4 based on a degree 8 Wilkinson polynomial over $I[0,1]$.

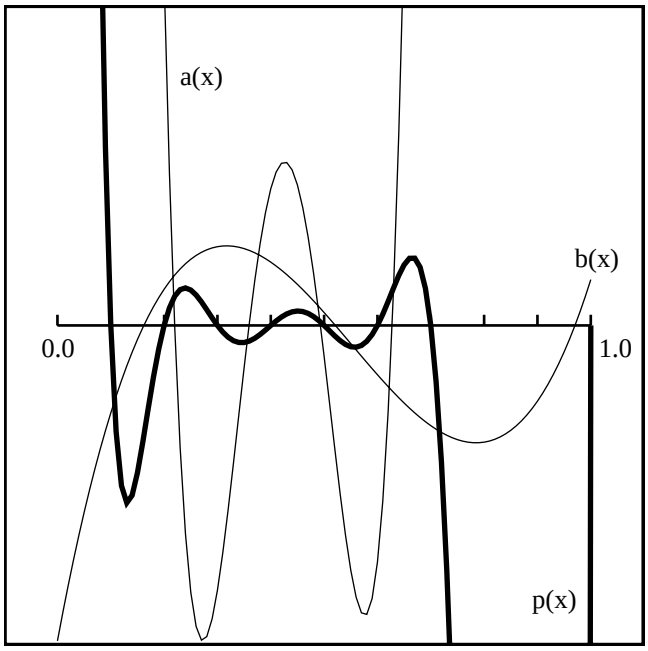

Figure 6.20: IPs of degree 3 and 4 based on a degree 8 polynomial over $I[0,1]$ with roots at $\{.1, .2, .3, .4, .5, .6, .7, 1.\}$.

Chapter 7

Numerical Results

This chapter presents results of empirical tests which compare implementations of the real root finding algorithms outlined in chapter 6. The algorithms were tested in double precision arithmetic on an IBM RS6000/350 workstation with floating-point machine constants as summarized in table 7.1. These constants are defined in the system file *\usr\include\float.h* [Pla92].

IBM RS6000 Series Machines		
Value	Constant	Description
2	FLT_RADIX	Machine Base
53	DBL_MANT_DIG	Mantissa Bits
2.2204460492503131E–16	DBL_EPSILON	Machine Tolerance
1.1102230246251565E–16	η	Machine Roundoff Unit
2.2250738585072014E–308	DBL_MIN	Machine Underflow Limit
1.7976931348623157E+308	DBL_MAX	Machine Overflow Limit

Table 7.1: Machine Floating-Point Constants

The following Bernstein form polynomial root finding algorithms are included in the benchmarked results. These root finders are specifically designed to find all real roots in the unit interval.

BCOM [§ 6.3] is implemented in FORTRAN.

BCHA$_1$ [§ 6.1] is implemented in FORTRAN.

BCHA$_2$ [§ 6.2] is implemented in FORTRAN.

BCHA$_3$ [§ 6.2] is implemented in FORTRAN.

BCHA$_4$ [§ 6.2] is implemented in FORTRAN.

BCHI [§ 6.4] is implemented in FORTRAN.

ROCKWOOD [§§ 2.6.1] is implemented in FORTRAN based on the pseudo code outline presented in [Roc89].

LANE–SCHNEIDER [§§ 2.6.1] is implemented in C based on the code in [Sch90a] with some simplification and corrections.

Routines which are written for power basis polynomials can also solve for Bernstein roots by using the pseudo basis conversion [§ 5.8]. This conversion does not induce any appreciable error. The following algorithms, included in the benchmarked results, compute all real and complex roots of a polynomial in power basis.

RPOLY [§§ 2.4] is the self-contained Jenkins–Traub FORTRAN program taken directly from [Jen75] with its intrinsic functions updated to accommodate double precision variables.

LAGUERRE [§§ 2.4] is written in FORTRAN as found in [PFTV86], with intrinsic functions changed to accommodate double precision.

DUNAWAY [§§ 2.4] is written in FORTRAN, taken directly from [Dun72].

MADSEN–REID [§§ 2.4] is written in FORTRAN, copied directly from [MR75]. Minor modification was performed to eliminate occasional floating-point overflows.

The following algorithms, included in the benchmarked results, compute only the real roots of a power basis polynomial.

CLOSED FORM [§ 5.9, §§ 2.3.1], the closed–form solutions to polynomials of degree two, three, and four, are implemented in both FORTRAN and C using concepts primarily found in [Lod90, PFTV90] with some constructions noted in [Sch90b]. Only real roots are computed.

EXCLUSION [§ 2.5.4] was generously provided by its authors. It is written in C. The basic algorithm returns a set of disjoint intervals which each contain exactly one root. We coupled it with the same MODIFIED REGULA FALSI C algorithm used in the STURM (HOOK–MCAREE) algorithm.

SIPS [§ 6.5] is written in C and uses the the C versions of CLOSED FORM. Its MODIFIED REGULA FALSI algorithm is the same one used by BCOM, BCHA_{2-4}, and BCHI modified to accommodate power form evaluations and converted to C.

STURM (HOOK–MCAREE) [§§ 2.5.1] is written in C, taken directly from [HM90].

The algorithms were polished and debugged sufficient enough to provide a rough indication of their relative performance. Only BCOM, RPOLY, and LAGUERRE performed robustly in almost every case. We suspect that the implementation of each of the other algorithms can be improved to handle cases of polynomials simple roots reliably.

Relative differences in execution time of less than a factor of, say, three should not be attributed much significance. These relative performance measures are rather sensitive to what machine the tests are being run on, and what compiler or compiler options are being used, as well as which set of polynomials is being tested. Furthermore, we did not make much effort to optimize the code.

7.1 Test Problems and Results for Distinct Real Roots

Number of roots in $[0,1]$	1	2	5
BCOM	1	1.1	2.9
BCHA_1	1.9	1.7	4.2
BCHA_2	2.3	2.3	7.2
BCHI	1.2	1.6	3.4
ROCKWOOD	1	1.1	4
LANE–SCHNEIDER	4.3		6.8
RPOLY	3.2	2	1.5
EXCLUSION	15	11	20
SIPS	2.4	1	1
LAGUERRE	5.6	3	3
DUNAWAY	83	46	
STURM (HOOK–MCAREE)	3.2	1.7	3.5
Average Unit Time (sec)	.92e-3	.43e-2	.16e-2

Table 7.2: Relative Execution Times for Degree Five Polynomials

Number of roots in $[0,1]$	1	2	5	10
BCOM	1	1	1	1.4
BCHA_1	2.5	2.3	2.0	2.2
BCHA_2	4.1	4.4	4.4	4.2
BCHA_3	3.7	—	—	—
BCHA_4	3.0	50	—	—
BCHI	1.5	2.8	2.3	2.7
ROCKWOOD	1.6	1.7	2.0	2.5
LANE–SCHNEIDER	—	—	4.8	3.5
RPOLY	9.0	4.8	1.9	1
LAGUERRE	16	7.8	2.5	1.3
DUNAWAY	170	98	45	21
STURM (HOOK–MCAREE)	109	71	28	16
Average Unit Time (sec)	.12e-2	. .22e-2	. .60e-2	. .19e-1

Table 7.3: Relative Execution Times for Degree Ten Polynomials

Number of roots in $[0,1]$	1	2	3	10	20
BCOM	1	1	1	1	1.4
BCHA_1	1.64	4.1	4.3	3.1	4.1
BCHI	1.7	4	4.1	2.7	5.6
ROCKWOOD	2.7	2.8	3.3	3.2	5.9
RPOLY	19	11	7.7	1.3	1
EXCLUSION	97	50	36	—	—
LAGUERRE	45	23	15	2.5	1.7
Average Unit Time (sec)	.34e-2	.72e-2	.17e-1	.44e-1	.21e-1

Table 7.4: Relative Execution Times for Degree Twenty Polynomials

Degree	2	3	4	5	10	15	20	25
BCOM	29	25	17	1.4	1.38	1.44	1.55	1.56
BCHA_1	21	22	17	1.6	2	2.7	3.5	4.4
BCHA_2	21	17	24	2.5	3.4	4.8	6.55	8.3
BCHA_3	—	17	12	1.8	2.3	3.2	4.73	X
BCHA_4	—	—	12	1.3	2.1	3.5	X	X
BCHI	16	16	17	1.7	2.25	3.8	4.73	5.9
ROCKWOOD	8	18	14	15	1.6	2.5	3.8	4.82
LANE–SCHNEIDER	131	193	79	10	145	24	84	10.4
CLOSED FORM	1	1	1	—	—	—	—	—
RPOLY	2	2	8.3	1	1	1	1	1
EXCLUSION	71	180	80	9.7	11.6	15	18.4	8.7
SIPS	1	3.2	2.8	1.27	—	—	—	—
LAGUERRE	12	13	9	1.01	1.25	1.25	1.45	1.44
DUNAWAY	220	127	67	11	94	25	73	11
MADSEN–REID	13	16	13	1.2	1.13	1.94	8.1	X
STURM (HOOK–MCAREE)	11	13	127	41	27	31	37	42
Average Unit Time (sec)	.25e-5	.67e-5	.15e-4	.22e-3	.40e-3	.16e-2	.22e-2	.32e-2

Table 7.5: Relative Execution Times for Wilkinson Polynomials in $I[0,1]$

Number of roots in $[0,1]$	0	1	2
Closed Form	2.5	1.4	1
Bcom	16	14	26
Bcha_1	9	24	20
Bcha_2	6	12	13
Bcha_3	8	12	13
Bcha_4	7	11	13
Bchi	10	15	20
Rockwood	5	10	15
Lane–Schneider	1	60	97
Rpoly	5.5	2	2
Exclusion	151	102	133
Sips	4.5	1	1.4
Laguerre	30	13	11
Dunaway	311	129	109
Sturm (Hook–McAree)	46	15	14
Average Unit Time (sec)	.20e-4	.50e-4	.29e-3

Table 7.6: Relative Execution Times for Degree Two Polynomials

Number of roots in $[0,1]$	0	1	2	3
Closed Form	16	1	1	1
Bcom	35	5	12	17
Bcha_1	16	9	16	20
Bcha_2	16	10	14	28
Bcha_3	14	6	11	
Bchi	24	5	14	14
Rockwood	10	4		18
Lane–Schneider	1	22	50	54
Rpoly	112	7	9	6.4
Exclusion	507	51	77	101
Sips	12	7	13	13
Laguerre	160	12	11	12
Dunaway	3255	205	97	X
Sturm (Hook–McAree)	147	10	11	10
Average Unit Time (sec)	.10e-4	.15e-3	.12e-3	.74e-3

Table 7.7: Relative Execution Times for Degree Three Polynomials

Number of roots in $[0,1]$	1	2	3	4
Closed Form	1	1	1	1
Bcom	2	5	7.6	8.8
Bcha_1	4	6	10	14
Bcha_2	5	9	15	22
Bcha_3	5	7	10	
Bcha_4	20	5	7	
Bchi	3	6	10	9
Rockwood	2	4	8	12
Lane–Schneider	11	19	32	22
Rpoly	6	5	7	4.5
Exclusion	31	36	52	62
Sips	5	6	8	6
Laguerre46.5	10	7	8	8
Dunaway	143	92	71	
Sturm (Hook–McAree)	6	6	7	32
Average Unit Time (sec)	.34e-3	.39e-3	.94e-2	.80e-3

Table 7.8: Relative Execution Times for Degree Four Polynomials

Number of roots in $[0,1]$	2	3	5
Bcom	1	1	1
Rpoly	8	6	3.5
Laguerre	20	14	8
Average Unit Time (sec)	.42e-2	.64e-2	.12e-1

Table 7.9: Relative Execution Times for Degree 20 with Clustered Roots

Number of roots in $[0,1]$	2	3	4	5
Bcom	1	1	1	1
Rpoly	3.8	2.7	1.8	1.4
Laguerre	7.4	5.1	3.4	2.3
Average Unit Time (sec) .26e-2		.37e-2	.53e-2	.68e-2

Table 7.10: Relative Execution Times for Degree 10 with Clustered Roots

7.2 Discussion

For polynomials of degree two, three, and four, the CLOSED FORM algorithms are clearly fastest. For degree five through nine, BCOM and ROCKWOOD appear to be fastest, though SIPS works well if there are several real roots, especially if they are clustered.

For degree ten and higher, BCOM appears to be fastest when the number of real roots in $[0, 1]$ is small. If the number of real roots approaches the degree of the polynomial, RPOLY is the best of the algorithms we tested. We ran some sample tests for higher degree polynomials, and BCOM was almost always 10–30 times faster than RPOLY for 1–3 real roots in $[0, 1]$.

BCOM also is capable of handling multiple roots more intelligently than the other Bernstein root finders that we implemented, since it engages the roots of derivatives and performs a running error analysis. Furthermore, the algorithm can probably be sped up somewhat by performing the running error analysis only on the final few refinement iterations.

For degree greater than ten, BCOM can outperform RPOLY because it typically concerns itself with a small subset of the roots that RPOLY computes. BCOM runs faster than other Bernstein root finders such as ROCKWOOD, BCHA_1, and LANE–SCHNEIDER largely because it refines roots using the $O(n)$ Horner evaluation algorithm rather than the $O(n^2)$ de Casteljau algorithm.

Chapter 8

Conclusions

This study was motivated by a recurring problem in practical computer–aided design and computer graphics applications, namely, the determination of the real roots of a polynomial on a finite interval. The widespread use of the Bézier representation for curves and surfaces, and the inherent numerical stability of the Bernstein polynomial basis, provides a fruitful context for re–examination of this classical problem.

We have presented and analyzed a variety of algorithms that take advantage of the intuitive geometric properties and constructions associated with the Bernstein form. These algorithms are simple in implementation, and also compare favorably in performance with established power–form root solvers. In instances involving high–degree polynomials with only a few real roots on the domain of interest, which are typical of cases encountered in CAD applications, the Bernstein–form solvers can in fact be significantly faster than the state–of–the–art RPOLY power–form solver developed by Jenkins and Traub (which determines both real and complex roots).

The use of Bernstein–form solvers in the geometric processing of free–form curves and surfaces obviates the need to perform explicit Bernstein–to–power basis conversions, which can be numerically unstable (note also that the pseudo–basis conversion allows the roots of Bernstein–form polynomials to be found using power–form solvers without loss of accuracy).

In addition to taking advantage of the superior numerical stability of the Bernstein form defined over the nominal interval $t \in [0,1]$, as compared to the power form about $t = 0$, we showed that the monotonically–improving condition of Bernstein representations with respect to subdivision yields a practical "preconditioning strategy" for computing the real roots of ill–conditioned high degree polynomials.

Specifically, when constructing the Bernstein coefficients of, say, a polynomial whose roots represent the intersections of two Bézier curves, the relative coefficient errors incurred in the "construction" process are found to be fairly independent of the size of the chosen parameter interval, whereas even a small reduction in the Bernstein interval width can result in dramatic improvements in the root condition numbers. Thus, by constructing Bernstein representations on a few subintervals, rather than the full domain $[0,1]$, intersection points can be computed to very high accuracy.

Bibliography

Bibliography

[Act70] Forman S. Acton. *Numerical Methods That Work*. Harper and Row Publications, Inc., New York, 1970.

[AH83] Gotz Alefeld and Jurgen Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983.

[Ait26] A.C. Aitken. On Bernoulli's numerical solution of algebraic equations. *Proceedings of the Royal Society of Edinburgh*, 46:289–305, 1926.

[Alt83] Rene Alt. Minimizing the error propagation in the integration of ordinary differential equations. In R. Stepleman et al., editor, *Scientific Computing*, IMACS Transactions on Scientific Computation, pages 231–235. North-Holland, Amsterdam, 1983.

[Alt86] R. Alt. The use of the CESTAC method in the parallel computation of roots of polynomials. In Robert Vichnevetsky and Jean Vignes, editors, *Numerical Mathematics and Applications*, IMACS Transactions on Scientific Computation, pages 3–9. North-Holland, Amsterdam, 1986.

[BC86] Marie-Christine Brunet and Francoise Chatelin. CESTAC, a tool for stochastic round-off error analysis in scientific computing. In Robert Vichnevetsky and Jean Vignes, editors, *Numerical Mathematics and Applications*, IMACS Transactions on Scientific Computation - 85, pages 11–20. North-Holland, Amsterdam, 1986.

[BDM84] C. Bouville, J.L. Dubois, and I. Marchal. Generating high quality pictures by ray tracing. In BØ, K. and Tucker, H.A., editor, *Eurographics'84*, pages 161–177, Copenhagen, September 1984. North-Holland.

[Bel90] R. Bell. IBM RISC System/6000 performace tuning for numerically intensive FORTRAN and C programs. Technical Report Document GG24–3611, IBM International Technical Support Center, Poughkeepsie, NY, 1990.

[BFK84] Wolfgang Bohm, Gerald Farin, and Jurgen Kahmann. A survey of curve and surface methods in cagd. *Computer Aided Geometric Design*, 1:1–60, 1984.

[BFR81] Richard L. Burden, J. Douglas Faires, and Albert C. Reynolds. *Numerical Analysis*. Prindle, Weber & Schmidt, Boston, 2nd edition, 1981.

[Boc64] Maxime Bocher. *Introduction to Higher Algebra*. Dover Publications, Inc., New York, 1964.

[Bod49] E. Bodewig. On types of convergence and on the behavior of approximations in the neighborhood of a multiple root of an equation. *Quarterly of Applied Mathematics*, 7:325–333, 1949.

[Bor50] Samuel Borofsky. *Elementary Theory Of Equations*. The MacMillan Company, New York, 1950.

[Bor85] G.J. Borse. *Fortran 77 and Numerical Methods for Engineers*. PWS Engineering, Boston, 1985.

[BP60] William Snow Burnside and Arthur William Panton. *The Theory Of Equations: With an Introduction to the Theory of Binary Algebraic Forms*, volume 1. Dover Publications, Inc., New York, 7th edition, 1960.

[Bre71] R.P. Brent. An algorithm with guaranteed convergence for finding a zero of a function. *The Computer Journal*, 14(4):422–424, 1971.

[Bre73] Richard P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[Buc66] R.A. Buckingham. *Numerical Methods*. Sir Issac Pitman & Sons LTD., London, 1966.

[BV80] P. Bois and Jean Vignes. A software for evaluating local accuracy in the Fourier transform. *Mathematics and Computers in Simulation*, XXII:141–150, 1980.

[CA76] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descarte's rule of signs. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 272–275, 1976.

[Caj04] Florian Cajori. *An Introduction To The Modern Theory Of Equations*. The MacMillan Company, New York, 1904.

[CDM79] Harlan Crowder, Ron S. Dembo, and John M. Mulvey. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software*, 5(2):193–203, June 1979.

[CL76] George E. Collins and Rüdiger Loos. Polynomial real root isolation by differentiation. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 15–25, 1976.

[CL82] George E. Collins and Rüdiger Loos. Real zeros of polynomials. *Computing, Suppl.*, 4:83–94, 1982.

[CLR80] Elaine Cohen, Tom Lyche, and Richard F. Riesenfeld. Discrete B–splines and subdivision techniques in computer–aided geometric design and computer graphics. *Computer Graphics and Image Processing*, 14:87–111, 1980.

[CS66] G.T. Cargo and O. Shisha. The bernstein form of a polynomial. *Journal Of Research of the National Bureau of Standards - B, Mathematics and Mathematical Physics*, 70B(1):79–81, January–March 1966.

[Dav27] Charles Davison. *Higher Algebra: For Colleges and Secondary Schools*. University Press, Cambridge, 2nd edition, 1927.

[DB75] G. Dahlquist and Å. Björck. *Numerical Methods*. Prentice–Hall, Englewood Cliffs, NJ, 1975.

[DD88] M. Daniel and J.C. Daubisse. The numerical problem of using Bézier curves and surfaces in the power basis. *Computer Aided Geometric Design*, 6:121–128, 1988.

[Dic22] Eugene Leonard Dickson. *First Course In The Theory Of Equations*. John Wiley & Sons, Inc., New York, 1922.

[DM72] William S. Dorn and Daniel D. McCracken. *Numerical Methods with Fortran IV Case Studies*. John Wiley & Sons, Inc., New York, 1972.

[Dod78] Irving Allen Dodes. *Numerical Analysis for Computer Sciences*. North–Holland, New York, 1978.

[DS90] David Dobkin and Deborah Silver. Applied computational geometry: Towards robust solutions of basic problems. *Journal of Computer and System Sciences*, 40:70–87, 1990.

[Dun72] Donna K. Dunaway. *A Composite Algorithm for Finding Zeros of Real Polynomials*. PhD thesis, Southern Methodist University, August 21 1972. FORTRAN code included.

[Dun74] Donna K. Dunaway. Calculation of zeros of a real polynomial through factorization using Euclid's algorithm. *SIAM J. Numer. Anal.*, 11(6):1087–1104, December 1974.

[DY93] Jean-Pierre Dedieu and Jean-Claude Yakoubsohn. Computing the real roots of a polynomial by the exclusion algorithm. *Numerical Algorithms*, 4:1–24, 1993.

[ECS94] Academic careers for experimental computer scientists and engineers. *Communications of the ACM*, 37(4):87–90, 1994.

[Far86] R.T. Farouki. The characterization of parametric surface sections. *Computer Vision, Graphics, and Image Processing*, 33:209–236, 1986.

[Far91a] R.T. Farouki. Computing with barycentric polynomials. *The Mathemtical Intelligencer*, 13:61–69, 1991.

[Far91b] R.T. Farouki. On the stability of transformations between power and Bernstein polynomial forms. *Computer Aided Geometric Design*, 8:29–36, 1991.

[Fay83] J.-P. Faye. Stabilizing eigenvalue algorithms. In R. Stepleman et al., editor, *Scientific Computing*, IMACS Transactions on Scientific Computation, pages 245–249. North-Holland, Amsterdam, 1983.

[FJ94] R.T. Farouki and J.K. Johnstone. The bisector of a point and a plane parametric curve. *Computer Aided Geometric Design, to appear*, 11, 1994.

[FL85] M.R. Farmer and G. Loizou. Locating multiple zeros interactively. *Comp. & Maths with Appls.*, 11:595–603, 1985.

[FMM77] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[FN90a] R.T. Farouki and C. A. Neff. Analytic properties of plane offset curves & Algebraic properties of plane offset curves. *Computer Aided Geometric Design*, 7:89–99 & 100–127, 1990.

[FN90b] R.T. Farouki and C.A. Neff. On the numerical condition of Bernstein–Bézier subdivision processes. *Mathematics of Computation*, 55:637–647, 1990.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[FR88] R.T. Farouki and V.T. Rajan. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design*, 5:1–26, 1988.

[FS65] D.K. Faddeev and I.S. Sominskii. *Problems in Higher Algebra*. W. H. Freeman & Company, San Francisco, 1965.

[FV85] J.-P. Faye and J. Vignes. Stochastic approach of the permutation-perturbation method for round-off error analysis. *Applied Numerical Mathematics*, 1:349–362, 1985.

[Gar79] Irene Gargantini. The numerical stability of simultaneous iterations via square-rooting. *Comp. & Maths. with Appls.*, 5:25–31, 1979.

[GH72] Irene Gargantini and Peter Henrici. Circular arithmetic and the determination of polynomial zeros. *Numerische Mathematik*, 18:305–320, 1972.

[GH73] J.A. Grant and G.D. Hitchins. The solution of polynomial equations in interval arithmetic. *The Computer Journal*, 16(1):69–72, 1973.

[GK90] S. Ghaderpanah and S. Klasa. Polynomial scaling. *SIAM J. Numer. Anal.*, 27(1):117–135, February 1990.

[GL70] Donald I. Good and Ralph L. London. Computer interval arithmetic: Definition and proof of correct implementation. *Journal of the ACM*, 17(4):603–612, 1970.

[Gla89] Andrew S. Glassner. Simple numerical root-finding. *The Ray Tracing News*, 3(1):19–22, May 1989. C code included.

[Gra65] A.A. Grau. Modified Graeffe method. *Communications of the ACM*, 8(6):379–380, 1965.

[Gra89] Thomas A. Grandine. Computing zeros of spline functions. *Computer Aided Geometric Design*, 6:129–136, 1989.

[GSA84] Ronald N. Goldman, Thomas W. Sederberg, and David C. Anderson. Vector elimination: A technique for the implicitization, inversion, and intersection of planar parametric rational polynomial curves. *Computer Aided Geometric Design*, 1:327–356, 1984.

[Ham71] Richard W. Hamming. *Introduction to applied numerical analysis*. McGraw–Hill, New York, 1971.

[Han69] E. Hansen. *Topics In Interval Analysis*. Clarendon Press, Oxford, 1969.

[Han70] Richard J. Hanson. Automatic error bounds for real roots of polynomials having interval coefficients. *The Computer Journal*, 13(3):284–288, August 1970.

[Han78a] E. Hansen. A globally convergent interval method for computing and bounding real roots. *BIT*, 20:415–424, 1978.

[Han78b] E. Hansen. Interval forms of Newton's method. *Computing*, 20:153–163, 1978.

[Han83] Pat Hanrahan. Ray tracing algebraic surfaces. *Computer Graphics (SIGGRAPH'83 Proceedings)*, 17:83–90, July 1983.

[HE86] Don Herbison-Evans. Solving quartics and cubics for graphics. Technical Report CS-86-56, University of Waterloo, November 1986.

[Hei71] Lee E. Heindel. Integer arithmetic algorithms for polynomial real zero determination. *Journal of the ACM*, 18(4):533–548, 1971.

[Hen64] Peter Henrici. *Elements of Numerical Analysis*. John Wiley & Sons, New York, 1964.

[Hen70] Peter Henrici. Uniformly convergent algorithms for the simultaneous determination of all zeros of a polynomial. In Ortega J.M. and Rheinboldt, W.C, editor, *Proceedings of the Symposium on Numerical Solution of Nonlinear Problems*, Studies in Numerical Analysis 2, pages 1–8, Philedelphia, 1970. SIAM.

[Hen82] Peter Henrici. *Essentials of Numerical Analysis: With Pocket Calculator Demonstrations*. John Wiley & Sons, New York, 1982.

[HG83] E.R. Hansen and R.I. Greenberg. An interval Newton method. *Applied Mathematics and Computation*, 12:89–98, 1983.

[Hil74] F.B. Hildebrand. *Introduction To Numerical Analysis*. International Series in Pure and Applied Mathematics. McGraw-Hill Book Company, Inc., New York, 2nd edition, 1974.

[HM90] D.G. Hook and P.R. McAree. Using Sturm sequences to bracket real roots of polynomial equations. In Andrew S. Glassner, editor, *Graphics Gems*, pages 416–422,743–755. Academic Press, Inc., San Diego, CA, 1990. C code included.

[Hou53] A.S. Householder. *Principles Of Numerical Analysis*. International Series in Pure and Applied Mathematics. McGraw-Hill Book Company, Inc., New York, 1953.

[Hou70] A.S. Householder. *The Numerical Treatment Of A Single Nonlinear Equation*. McGraw-Hill Book Company, Inc., New York, 1970.

[HP77] Eldon Hansen and Merrell Patrick. A family of root finding methods. *Numer. Math.*, 27:257–269, 1977.

[Ign71] James P. Ignizio. On the establishment of standards for comparing algorithm performance. *Interfaces*, 2(1):8–11, November 1971.

[IK66] Eugene Isaacson and Herbert Bishop Keller. *Analysis of Numerical Methods*. John Wiley & Sons, Inc., New York, 1966.

[Jen75] M.A. Jenkins. Algorithm 493: Zeros of a real polynomial [c2]. *ACM Transactions on Mathematical Software*, 1(2):178–192, June 1975. FORTRAN code included.

[JT70] M.A. Jenkins and J.F. Traub. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM J. Numer. Anal.*, 7(4):545–570, 1970.

[JT74] Michael A. Jenkins and Joseph F. Traub. Principles for testing polynomial zerofinding programs. In *Proceedings of Mathematical Software II*, pages 84–107, West Lafayette, Ind., Purdue Univ., May 29-31 1974.

[JT75] M.A. Jenkins and J.F. Traub. Principles for testing polynomal zerofinding programs. *ACM Transactions on Mathematical Software*, 1(1):26–34, March 1975.

[Kaj82] James T. Kajiya. Ray tracing parametric patches. *Computer Graphics (SIGGRAPH'82 Proceedings)*, 23(2):245–254, July 1982.

[KL73] Toyohisa Kaneko and Bede Liu. On local roundoff errors in floatng point arithmetic. *Journal of the Association for Computing Machinery*, 20(3):391–398, 1973.

[KM81] Ulrich W. Kulisch and Willard L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, New York, 1981.

[Kur80] A. Kurosh. *Higher Algebra*. Mir Publishers, Moscow, 3rd edition, 1980.

[Lin41] Shih-nge Lin. A method of successive approximations of evaluating the real and complex roots of cubic and higher-order equations. *Journal of Mathematics and Physics*, 20:231–242, 1941.

[Lin43] Shih-nge Lin. A method for finding roots of algebraic equations. *Journal of Mathematics and Physics*, 20:60–77, 1943.

[LN79] Charles C. Lee and H.P. Niu. Determination of zeros of polynomials by synthetic division. *International Journal of Computer Mathematics*, 7:131–139, 1979.

[Lod90] Suresh Lodha. Computing real zeros of quartic and lower order polynomials in closed-form. IBM Research Report RC16056, IBM Research Division, T. J. Watson Research Center Yorktown Heights, NY 10598, October 1990.

[LR80] Jeffrey M. Lane and Richard F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Trans. Pattern Anal. Machine Intell.*, 2:35–46, 1980.

[LR81] Jeffrey M. Lane and Richard F. Riesenfeld. Bounds on a polynomial. *BIT*, 21:112–117, 1981.

[Mac54] Cyrus Colton MacDuffee. *Theory Of Equations*. John Wiley & Sons, Inc., New York, 1954.

[Mac63] Nathaniel Macon. *Numerical Analysis*. John Wiley & Sons, Inc., New York, 1963.

[Mad73] Kaj Madsen. A root-finding algorithm based on Newton's method. *BIT*, 13:71–75, 1973.

[Mar66] Morris Marden. *Geometry of Polynomials*. Mathematical Surveys Number III. American Mathematical Society, Providence, Rhode Island, 1966.

[Mat79] G.K. Matthew. An alternative to Euclid's algorithm. *Transactions of the ASME: Journal of Mechanical Design*, 101:582–586, October 1979.

[MB72] Ian Munro and Allan Borodin. Efficient evaluation of polynomial forms. *Journal of Computer and System Sciences*, 6:625–638, 1972.

[MC90] Denis Marchepoil and Patrick Chenin. Algorithmes de recherche de zéros dúne fonction de Bézier. Laboratoire de Modélisation et Calcul RR 834-I & M-, Institut National Polytechnique de Grenoble, Université Joseph Fourier Grenoble 1, Centre National de la Recherche Scientifique, Ecole Normale Supérieure de Lyon, November 1990.

[McN73] Samuel S. McNeary. *Introduction To Computational Methods For Students Of Calculus*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[Mer06] Mansfield Merriman. *The Solution Of Equations*. Mathematical Monographs No. 10. John Wiley & Sons, New York, 4th edition, 1906.

[Mil75a] Webb Miller. Computer search for numerical instability. *Journal of the Association for Computing Machinery*, 1(2):512–521, October 1975.

[Mil75b] Webb Miller. Software for roundoff analysis. *ACM Transactions on Mathematical Software*, 1(2):108–128, June 1975.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[Moo76] J.B. Moore. A consistently rapid algorithm for solving polynomial equations. *J. Inst. Maths. Applics*, 17:99–110, 1976.

[Moo77] J.B. Moore. A test for existence of solutions to nonlinear systems. *SIAM Journal of Numerical Analysis*, 14(4):611–615, 1977.

[Moo79] Ramon E. Moore. *Methods And Applications Of Interval Analysis*. SIAM Studies in Applied Mathematics. SIAM, Philadelphia, 1979.

[Mor83] J. Ll. Morris. *Computational Methods In Elementary Numerical Analysis*. John Wiley & Sons, Inc., New York, 1983.

[MP65] A.P. Mishina and I.V. Proskuryakov. *Higher Algebra: Linear Algebra, Polynomials, General Algebra*. Pergamon Press Ltd., Oxford, 1st English edition, 1965.

[MR75] K. Madsen and J.K. Reid. Fortran subroutines for finding polynomial zeros. Technical Report A.E.R.E. R.7986, Computer Science and Systems Division, AERE Harwell, Computer Science and Systems Division, A.E.R.E. Harwell, Didcot, Oxford, England, February 1975.

[Mul56] David E. Muller. A method for solving algebraic equations using an automatic computer. *Math. Tab., Wash.*, 10:208–215, 1956.

[Non84] T.R.F. Nonweiler. *Computational Mathematics: An Introduction to Numerical Approximation*. Ellis Horwood Limited, Chichester, 1984.

[Ost60] A.M. Ostrowski. *Solution of Equations and Systems of Equations*. Academic Press, New York, 1960.

[Pav82] Theo Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, MD, 1982.

[PC86] Chris Phillips and Barry Cornelius. *Computational Numerical Methods*. Ellis Horwood Limited, Chichester, 1986.

[Pen70] Ralph H. Pennington. *Computer Methods and Numerical Analysis*. Macmillian Co., Toronto, 1970.

[Pet81] M.S. Petkovic. On a generalization of the root iterations for polynomial complex zeros in circular interval arithmetic. *Computing*, 27:37–55, 1981.

[Pet89] Miodrag Petkovic. *Iterative Methods For Simultaneous Inclusion of Polynomial Zeros*. Lecture Notes in Mathematics #1387. Springer-Verlag, Berlin, 1989.

[PFTV86] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1986.

[PFTV90] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1990.

[Piz75] Stephen M. Pizer. *Numerical Computing And Mathematical Analysis*. Science Research Associates, Inc., Chicago, 1975.

[Pla92] P.J. Plauger. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[PW71] G. Peters and J.H. Wilkinson. Practical problems arising in the solutions of polynomial equations. *J. Inst. Maths Applies*, 8:16–35, 1971.

[PW83] Stephen M. Pizer and Victor L. Wallace. *To Compute Numerically: Concepts and Strategies*. Little, Brown & Company, Boston, 1983.

[Ric83] John R. Rice. *Numerical Methods, Software, and Analysis*. McGraw-Hill Book Company, New York, imsl reference edition, 1983.

[Riv70] T.J. Rivlin. Bound on a polynomial. *Journal of Research of the National Bureau of Standards - B, Mathematical Sciences*, 74B(1):47–54, January-March 1970.

[RKF88] V.T. Rajan, S.R. Klinkner, and R.T. Farouki. Root isolation and root approximation for polynomials in Bernstein form. IBM Research Report RC14224, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, N.Y. 10598, November 1988.

[RL71] J. Rokne and P. Lancaster. Complex interval arithmetic. *Communications of the ACM*, 14(2):111–112, February 1971.

[Roc89] Alyn Rockwood. Constant component intersections of parametrically defined curves. Technical report, Silicon Graphics Computer Systems, 1989.

[Roc90] Alyn Rockwood. Accurate display of tensor product isosurfaces,. *Proc IEEE Visualization '90 Conf.*, 1990.

[Rok75] J. Rokne. Reducing the degree of an interval polynomial. *Computing*, 14:5–14, 1975.

[Rok77] J. Rokne. Bounds for an interval polynomial. *Computing*, 18:225–240, 1977.

[RR78] Anthony Ralston and Philip Rabinowitz. *A First Course In Numerical Analysis*. McGraw-Hill Book Company, New York, 2nd edition, 1978.

[Rum79] Siegfried M. Rump. Polynomial minimum root separation. *Mathematics of Computation*, 33(145):327–336, January 1979.

[SA84] T.W. Sederberg and D.C. Anderson. Ray tracing of steiner patches. *Computer Graphics (SIGGRAPH'84 Proceedings)*, 18:159–164, 1984.

[SAG85] T.W. Sederberg, David C. Anderson, and Ronald N. Goldman. Implicitization, inversion, and intersection of planar rational cubic curves. *Computer Vision, Graphics, and Image Processing*, 31:89–102, 1985.

[Sal85] George Salmon. *Modern Higher Algebra*. Chelsea Publishing Company, New York, 5th edition, 1885.

[SC94] Thomas W. Sederberg and Geng-Zhe Chang. Isolator polynomials. In Chandrajit L. Bajaj, editor, *Algebraic Geometry and Its Applications*, chapter 32, pages 507–512. Springer-Verlag, New York, 1994.

[Sch90a] Philip J. Schneider. A Bézier curve-based root-finder. In Andrew S. Glassner, editor, *Graphics Gems*, pages 408–415,787–797. Academic Press, Inc., San Diego, CA, 1990. C code included.

[Sch90b] Jochen Schwarze. Cubic and quartic roots. In Andrew S. Glassner, editor, *Graphics Gems*, pages 404–407,738–742. Academic Press, Inc., San Diego, CA, 1990. C code included.

[Sed85] Thomas W. Sederberg. Piecewise algebraic surface patches. *Computer Aided Geometric Design*, 2:53–59, 1985.

[Sed89] Thomas W. Sederberg. Algorithm for algebraic curve intersection. Computer-Aided Design, 21(9):547–554, 1989.

[Sed93] Thomas W. Sederberg, 1993. private communication.

[SF92] Thomas W. Sederberg and Rida T. Farouki. Approximation by interval Bézier curves. *IEEE CG&A*, 12(5):87–95, September 1992.

[SN90a] Thomas W. Sederberg and Tomoyuki Nishita. Curve intersection using Bézier clipping. Computer-Aided Design, 22(9):538–549, 1990.

[SN90b] T.W. Sederberg and Tomoyuki Nishita. Ray tracing trimmed rational surface patches. *Computer Graphics (SIGGRAPH'90 Proceedings)*, 24:337–345, 1990.

[SP86] Thomas W. Sederberg and Scott R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, January/February 1986.

[SR77] C.E. Schmidt and L.R. Rabiner. A study of techniques for finding the zeros of linear phase fir digital filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 25(1):96–98, 1977.

[SSd86] Thomas W. Sederberg, Melvin R. Spencer, and Carl de Boor. Real root approximation of polynomials in Bernstein form. draft copy, December 1986.

[Sta70] Peter A. Stark. *Introduction to Numerical Analysis.* Macmillan Publishing Co., Inc., New York, 1970.

[Ste73] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[Ste74] Pat H. Sterbenz. *Floating-Point Computation.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.

[SWZ89] Thomas W. Sederberg, Scott C. White, and Alan K. Zundel. Fat arcs: a bounding region with cubic convergence. *Computer Aided Geometric Design*, 6:205–218, 1989.

[Tol83] Pierre Tolla. Linear and non-linear programing software validity. *Mathematics and Computers in Simulation*, XXV:39–42, 1983.

[Tra64] J.F. Traub. *Iterative Methods For The Solution Of Equations.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1964.

[Tur52] H.W. Turnbull. *Theory Of Equations*. Oliver & Boyd Ltd., Edinburgh, 5th edition, 1952.

[Usp48] J.V. Uspensky. *Theory Of Equations*. McGraw-Hill, New York, 1948.

[Van70] H. Van de Vel. A note on the automatic pretreatment of polynomials. *The Computer Journal*, 13(3):289–290, 1970.

[Vig78] Jean Vignes. New methods for evaluating the validity of the results of mathematical computations. *Mathematics and Computers in Simulation*, XX:227–249, 1978.

[Vig84] Jean Vignes. An efficient implementation of optimization algorithms. *Mathematics and Computers in Simulation*, XXVI:243–256, 1984.

[Vig88] Jean Vignes. Review on stochastic approach to round-off error analysis. *Mathematics and Computers in Simulation*, 30:481–491, 1988.

[VL74] Jean Vignes and M. La Porte. Error analysis in computing. *Information Processing*, 74:610–614, 1974.

[Wes81] David H. West. Computer-assisted magnet shimming. *Review of Scientific Instruments*, 52(12):1910–1912, 1981.

[Wij84] J.J. Van Wijk. Ray tracing objects defined by sweeping a sphere. In BØ, K. and Tucker, H.A., editor, *Eurographics'84*, pages 73–82, Copenhagen, September 1984. North-Holland.

[Wil59] J.H. Wilkinson. The evaluation of the zeros of ill–conditioned polynomials, Parts I & II. *Numer. Math.*, 1:150–166 & 167–180, 1959.

[Wil60] J.H. Wilkinson. Error analysis of floating point computation. *Numer. Math.*, 2:319–340, 1960.

[Wil63] J.H. Wilkinson. *Rounding Errors In Algebraic Processes*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1963.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, 1991.

[Zun89] Alan K. Zundel. Scan line rendering of algebraic surfaces and half spaces. Master of Science, Brigham Young University, August 1989.

Polynomial Real Root Finding in Bernstein Form

Melvin R. Spencer

Department of Civil Engineering

Ph. D. Degree, August 1994

ABSTRACT

This dissertation presents pre-conditioning, isolation, and approximation concepts pertaining to finding the real roots of a polynomial in Bernstein from over a specified domain. Subdivision of a polynomial into smaller intervals prior to coefficient construction significantly improves the accuracy of the approximated roots as compared to "a posteriori" subdivision after the coefficients are generated.

Real root isolation and approximation strategies are presented that exploit various properties of the Bernstein representation to compute the real roots of polynomials in Bernstein form. The numerical performance of these strategies is compared with other existing Bernstein and power form polynomial real root-finding strategies.

COMMITTEE APPROVAL:

Thomas W. Sederberg, Committee Chairman

Norman L. Jones, Committee Member

Rida T. Farouki, Committee Member

S. Olani Durrant, Department Chairman