Computation of the Real Solutions of Nonlinear Polynomial Systems

Evan C. Sherbrooke

Nicholas M. Patrikalakis

Massachusetts Institute of Technology
Department of Ocean Engineering
Design Laboratory, Cambridge, MA 02139-9910, USA

MIT Ocean Engineering Design Laboratory Memorandum 91–12

October 30, 1991
Revised: June 3, 1992

Abstract

A fundamental problem in computer aided design is the efficient computation of all real roots of a system of nonlinear polynomial equations in n variables which lie within an n-dimensional box. Such problems often occur in computing intersections, distance functions, and extrema, as well as in engineering design. In this paper, we present two techniques designed to solve such problems, which rely on representation of polynomials in the multivariate Bernstein basis and subdivision. In order to isolate all of the roots within the given domain, each method uses a different scheme for constructing a series of bounding boxes; the first method projects control polyhedra onto a set of coordinate planes, and the second employs linear optimization. We also examine in detail the local convergence properties of the two methods, proving that the former is quadratically convergent for $n = 1$ and linearly convergent for $n > 1$, while the latter is quadratically convergent for all n. Worst-case complexity analysis, as well as analysis of actual running times are performed. Finally, we investigate the implementation of both techniques in exact rational arithmetic.

Keywords: CAD, CAGD, CAM, geometric modeling, solid modeling, intersections, distance computation, engineering design.

1 Introduction

A fundamental problem in computer-aided design and manufacturing is the efficient computation of all real solutions to a system of nonlinear polynomial equations within some finite domain. This problem arises in many different applications, as varied as they are numerous. For example, in computer-aided design it is often necessary to identify all characteristic points of an intersection curve between two surfaces, in order to trace out all of the branches of the curve; if the surfaces are polynomial (or, more generally, piecewise rational polynomial), we must solve a system of nonlinear polynomial equations in order to identify these points. In the area of feature recognition, computing the medial axis transform requires the determination of branch points, which can frequently be formulated as the solution set of a polynomial system. This problem also arises in planning robotic motions, in arranging distributive systems in complex mechanical platforms, and in many other areas of computer-aided engineering.

In recent years, geometric design systems which rely upon piecewise rational polynomial entities to represent geometry have proliferated. As our computers have increased in power, so have the sizes of the problems that we wish to solve. We seek, therefore, a method of solution which is effective regardless of input but whose execution time does not grow unacceptably with the size of the problem.

In this paper, we describe two techniques for eliciting all real solutions to a system of n nonlinear polynomial equations in n unknowns over an n-dimensional rectangular domain. Section 2 briefly reviews some of the prior research related to this topic. The notation we will use is introduced in section 3, and section 4 gives the formulation of the problem. Section 5 describes and analyzes the projected-polyhedron (PP) technique. In section 6, we introduce a new technique based on linear programming (LP), and show that this method exhibits quadratic convergence irrespective of n. Some examples comparing the performance of these two approaches follow in section 7, along with a brief survey of some of the applications of this technique. A few concluding remarks concerning the use of exact rational arithmetic are given in section 8, along with recommendations for future research. Two appendices are also included: appendix A discusses conversion from the multivariate monomial (power) basis to the multivariate Bernstein basis, and appendix B discusses De Casteljau subdivision applied to multivariate polynomials in the Bernstein basis.

2 Literature Review

In recent CAD-related research, three classes of methods for the computation of solutions of nonlinear polynomial systems have been favored: algebraic techniques, homotopy, and subdivision. These methods may be classified as *global* because they are designed to compute all roots in some area of interest. There also exist a number of *local* numerical techniques which employ some variation of Newton-Raphson iteration or numerical optimization [7]. These methods are used in CAD applications requiring high accuracy because they are efficient (usually exhibiting quadratic convergence rates close to simple roots) and are straightforward to program. However, they typically require good initial approximations to roots; such approximations are usually obtained through some sort of global search like sampling, a process which cannot provide full assurance that all roots have been found. This lack of robustness makes the development of efficient and stable global techniques desirable; in what follows, we briefly review three classes of global methods.

The computation of all complex roots of nonlinear polynomial systems has typically been approached with *algebraic geometry techniques* like elimination or Groebner basis methods [4] [3]. These methods have many advantages; they are theoretically elegant, guaranteed to find all complex roots of a system irrespective of the dimensionality of the solution set, and well-suited for implementation in symbolic mathematical systems [30]. However, they suffer from numerical instability, making implementation in floating point arithmetic difficult. Furthermore, they are inefficient in memory and processing time requirements, making them unattractive for systems only moderate in degree or dimensionality. There is also a philosophical problem with these techniques: they frequently give us much more information than we need. As we shall see in section 7, many CAD applications only require those roots which lie in some real n-dimensional box.

The second category of methods is the class of *homotopy techniques* [33] [12]. These methods may be used to find all complex solutions of a nonlinear polynomial system if the number of roots is finite. Unfortunately, investigation of such methods indicates that they also tend to be numerically ill-conditioned. If we try to get around this problem by implementing the algorithm in exact rational arithmetic, we end up with enormous memory requirements because we have to solve large systems of complex initial value problems. And homotopy suffers from the same philosophical problem as algebraic geometry techniques — it simply provides more information than we need.

The third class, the category of *subdivision-based techniques*, is the class to which both the Projected-

Polyhedron and Linear Programming methods described in this paper belong. Lane and Riesenfeld [18] investigated the application of binary subdivision and the variation diminishing property of polynomials in the Bernstein basis to eliciting the real roots and extrema of a polynomial within an interval. Boehm [2] and Cohen et al. [5] extended this idea to general nonuniform subdivision of B-splines. Geisow [14] was among the first to develop subdivision algorithms to intersect planar algebraic curves expressed in the tensor-product and barycentric Bernstein bases and apply them to surface intersections. Formulation of a class of geometric problems in more than one dimension, requiring the solution of nonlinear piecewise polynomial systems expressed in terms of B-splines, was made by Dokken[8], who suggested a solution approach based on [5]. Subdivision techniques have also more recently been used in a wide variety of intersection problems for geometric modeling. Sederberg [28] developed an adaptive subdivision algorithm which can be used to intersect two planar algebraic curves expressed in the barycentric Bernstein basis. Patrikalakis, Prakash, and Kriezis [24] [23] [17] investigated the use of subdivision of algebraic curves in intersecting an implicit algebraic surface with a rational polynomial surface. Their method relies on the computation of real characteristic points of an algebraic curve represented in the Bernstein basis, which typically involves intersecting two or three algebraic curves by repeated adaptive subdivision and minimization. Minimization is used to increase the precision of the root quadratically; alternatively, Newton-Raphson iteration, which is also quadratically convergent, can be used to refine the root. Nishita et al. [21] developed an adaptive subdivision technique to intersect rays with trimmed rational polynomial surface patches, also recasting the problem as the intersection of two algebraic curves expressed in the Bernstein basis. Sederberg and Nishita [29] extended this method to intersect parametric curves with parametric surfaces. Vafiadou and Patrikalakis [31] employed an early two-dimensional form of the Projected-Polyhedron algorithm coupled with minimization to ray-trace offset surfaces. In this paper, we will study the n-dimensional extension of the Projected-Polyhedron algorithm in detail, along with the related Linear Programming approach.

It should be noted that subdivision techniques themselves have a number of disadvantages. They are not as general as algebraic methods, since they are designed for zero-dimensional solution sets within an n-dimensional subset of Euclidean space. Furthermore, although the chances that all roots have been found increase as the resolution tolerance is lowered, there is no certainty that each root has been extracted. Lastly, subdivision techniques provide no explicit information about root multiplicities without additional computation. However, despite these drawbacks, their speed and stability make them attractive as root-finding schemes.

The subdivision methods reviewed above belong to the general class of interval-based methods, although they have a geometric character. Interval arithmetic techniques, primarily interval extension of Newton methods coupled with bisection to ensure convergence, have also been an active research topic in recent years [15]. An extensive bibliography and literature survey on this topic may be found in [20] [1]. The relative merits of geometrically-based subdivision methods and Newton-based interval techniques are a subject of further study.

3 Notation

Working in n dimensions tends to be a notational headache, so we will begin by introducing the notation and definitions used in this paper to avoid any confusion in subsequent sections. First, let us state the conventions we will use in naming variables:

1. The lower-case letters i, j, k, l, m, and n denote nonnegative integers.
2. The lower-case letters f, g, and h denote real-valued functions.
3. Other lower-case letters, such as x and y, denote real numbers.
4. Lower-case letters in boldface, such as $\mathbf{x}$ and $\mathbf{y}$ will denote vectors of real numbers, or points. It should be clear from the context what the dimension of any given vector is. When we work with the components of the vector $\mathbf{x}$, we will assume that x_i is the ith component of $\mathbf{x}$.

5. Upper-case boldface letters such as $\mathbf{F}$ denote vector-valued functions. The ith component of $\mathbf{F}$ is denoted by $\mathbf{F}_i$.
6. Upper-case letters other than those between I and N inclusive (which are used for multi-indices, defined below) denote sets or matrices.
7. To avoid confusion, some variables may have superscripts instead of or in addition to subscripts. For example, the matrix $W^{(k)}$ is superscripted so that we can refer to the element in the ith row, jth column as $w_{ij}^{(k)}$, rather than as the ambiguous w_{ijk}.
8. $[a, b]$ denotes set of all real x such that $a \le x \le b$.
9. $\mathbf{R}$ denotes the real numbers; $\mathbf{R}^n$ denotes the set of points in n-dimensional Euclidean space.

The only additions to these naming conventions are given in the definitions below.

Definition 3.1 *The* Euclidean norm $|\mathbf{x}|$ *of a vector* $\mathbf{x}$ *is defined to be the quantity* $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$.

Definition 3.2 *A* multi-index I *is an ordered n-tuple of nonnegative integers* $(i_1, i_2, \ldots, i_n)$. *We say that I is* bounded *by the multi-index* $M = (m_1, m_2, \ldots, m_n)$ *if* $0 \le i_1 \le m_1, 0 \le i_2 \le m_2, \ldots, 0 \le i_n \le m_n$.

We will use multi-indices as a shorthand for subscripting multi-dimensional arrays. For example, if $I = (2, 3)$, then w_I is equivalent to w_{23}, the element of the matrix $\{w_{ij}\}$ at row 2, column 3.

We will use the notation $\sum_I^M$, where M is a bounding multi-index, to indicate that a sum is to be taken over all possible multi-indices I which are bounded by M. For example, if $M = (1, 1)$, then $\sum_I^M w_I \equiv w_{00} + w_{01} + w_{10} + w_{11}$.

Definition 3.3 *The ith* Bernstein polynomial *of degree m is defined by*

$$b_{i,m}(u) = \binom{m}{i} u^i (1-u)^{m-i} \tag{1}$$

The Bernstein multinomial *determined by the multi-index* $I = (i_1, i_2, \ldots, i_n)$ *and the bounding multi-index* $M = (m_1, m_2, \ldots, m_n)$ *is defined by*

$$B_{I,M}(\mathbf{u}) = b_{i_1,m_1}(u_1) b_{i_2,m_2}(u_2) \ldots b_{i_n,m_n}(u_n) \tag{2}$$

where $\mathbf{u} = (u_1, u_2, \ldots, u_n) \in \mathbf{R}^n$.

Definition 3.4 *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be an element of* $\mathbf{R}^n$, *and let* $f : \mathbf{R}^n \to \mathbf{R}$ *be a function of* $\mathbf{x}$. *The* graph *of f is the function* $\mathbf{G}_f : \mathbf{R}^n \to \mathbf{R}^{n+1}$ *defined by*

$$\mathbf{G}_f(\mathbf{x}) = (\mathbf{x}, f(\mathbf{x})). \tag{3}$$

Definition 3.5 *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be an element of* $\mathbf{R}^n$. *Then the ith* projection map $\pi_i : \mathbf{R}^n \to \mathbf{R}$ *is defined by*

$$\pi_i(\mathbf{x}) = x_i. \tag{4}$$

Definition 3.6 *Suppose P is a set of m points of* $\mathbf{R}^n$. *Let $C(P)$ denote the set of all $\mathbf{x}$ such that*

$$\mathbf{x} = \sum_{i=1}^{m} c_i \mathbf{p}_i \tag{5}$$

where for each i, $\mathbf{p}_i \in P$ *and* $c_i \ge 0$, *and* $\sum_{i=1}^m c_i = 1$. *Then $C(P)$ is called the convex hull of P.*

Definition 3.7 *Suppose $f_1, f_2, \ldots, f_n$ are functions of the parameters $x_1, x_2, \ldots, x_n$. Then the* Jacobian matrix *of the f_i, $\frac{\partial\{f_1,f_2,\ldots,f_n\}}{\partial\{x_1,x_2,\ldots,x_n\}}$, is a square, $n \times n$ matrix whose element at row i, column j, is equal to $\frac{\partial f_i}{\partial x_j}$.*

Definition 3.8 *Suppose $f(x)$ and $g(x)$ are both real-valued functions of one variable. We say that $f(x) = \mathcal{O}(g(x))$ if there exists a $\delta > 0$ and a constant $\lambda > 0$ such that for $|x| < \delta$, $|f(x)| \leq \lambda|g(x)|$.*

In some texts, particularly ones dealing with the theory of algorithms (for example, [6]), the $\mathcal{O}$ notation concerns behavior of f for large x rather than small x. We will be using that formulation in the analysis of algorithmic complexity. However, since in proofs of convergence rates we will be interested in the behavior of certain functions near zero, we have chosen to define the $\mathcal{O}$ notation in this way for them.

4 Formulation

Suppose we have a set of n functions of n variables $f_1, f_2, \ldots, f_n$, each of which is polynomial in the independent parameters $u_1, u_2, \ldots, u_n$. Let $m_i^{(k)}$ denote the degree in u_i of polynomial f_k, so that the multi-index $M^{(k)} = (m_1^{(k)}, m_2^{(k)}, \ldots, m_n^{(k)})$ describes all the degree information of f_k. Furthermore, suppose there is an n-dimensional rectangular subset of $\mathbf{R}^n$

$$S = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]. \tag{6}$$

We wish to find all points $\mathbf{u} = (u_1, u_2, \ldots, u_n) \in S$ such that

$$f_1(\mathbf{u}) = f_2(\mathbf{u}) = \ldots = f_n(\mathbf{u}) = \mathbf{0}. \tag{7}$$

By making the *affine parameter transformation* [9] $u_i = a_i + x_i(b_i - a_i)$ for each i between 1 and n inclusive, we simplify the problem to one of determining all $\mathbf{x} \in [0,1]^n$ such that

$$f_1(\mathbf{x}) = f_2(\mathbf{x}) = \ldots = f_n(\mathbf{x}) = \mathbf{0}. \tag{8}$$

Since all of the f_k are polynomial in each of the n independent parameters, a simple *change of basis* [9] allows us to express them in the multivariate Bernstein basis. In other words, for each f_k there exists an n-dimensional array of real coefficients $w_{i_1 i_2 \ldots i_n}^{(k)}$ such that for each $k \in \{1, \ldots, n\}$

$$f_k(\mathbf{x}) = \sum_{I}^{M^{(k)}} w_I^{(k)} B_{I,M^{(k)}}(\mathbf{x}). \tag{9}$$

Representation of algebraic and piecewise algebraic surfaces (i.e., for $n = 3$) in terms of tensor products of Bernstein polynomials or B-splines has been studied earlier by Patrikalakis and Kriezis [22]; equation (9) is simply an extension to n dimensions. Provided that conversion of the problem to the Bernstein basis is exact or sufficiently accurate, the use of the Bernstein basis in conjunction with subdivision is known to be numerically stable [10]. The conversion process itself is numerically ill-conditioned [11]; therefore, we recommend that the problem be formulated in the Bernstein basis from the very beginning. If this is impossible, the algorithm of appendix A may be used to convert from the multivariate power basis to the multivariate Bernstein basis.

The form of (8) is not sufficient to apply the two techniques we consider in this paper. Before we can use either method, we have to restate the problem as the intersection of the *graphs* of the f_k (each of which is a hypersurface in $\mathbf{R}^{n+1}$) and the hyperplane $x_{n+1} = 0$ of $\mathbf{R}^{n+1}$. This idea is designed to impart geometrical significance to the coefficients of the polynomials and to the solution process. A similar approach has been used extensively as the foundation of convex-hull based subdivision algorithms for intersection problems (see Section 2). This is also the same approach we take when we attempt to solve an equation by graphing. For example, suppose we want to find the solutions to $x^2 - 3x + 2 = 0$. We graph the curve on paper, noting that the curve intersects the x-axis at the solution points $x = 1$ and $x = 2$. Mathematically speaking, we are drawing the parametric curve $(x, x^2 - 3x + 2)$ and intersecting it with the line $y = 0$.

Let us take the same approach in rewriting equation (8). Out of each f_k we will build its graph $\mathbf{F}_k$ (see definition 3.4):

$$\begin{aligned}\mathbf{F}_k(\mathbf{x}) &= (x_1, x_2, \ldots, x_n, f_k(\mathbf{x})) \\ &= (\mathbf{x}, f_k(\mathbf{x})).\end{aligned} \tag{10}$$

Clearly, (8) is satisfied by the point $\mathbf{x}$ if and only if

$$\mathbf{F}_1(\mathbf{x}) = \mathbf{F}_2(\mathbf{x}) = \ldots = \mathbf{F}_n(\mathbf{x}) = (\mathbf{x}, 0). \tag{11}$$

As with the f_k, the $\mathbf{F}_k$ may be expressed in the multivariate Bernstein basis. It can be shown that ([9])

$$\sum_{i=0}^{m} \frac{i}{m} b_{i,m}(u) = u; \tag{12}$$

in other words, the monomial u can be expressed as the weighted sum of Bernstein polynomials with coefficients evenly spaced in the interval $[0, 1]$. Using equation (12) and the summation to unity property of the Bernstein basis (17), we obtain an equivalent expression for each of the x_j in equation (10):

$$x_j = \sum_{I}^{M^{(k)}} \frac{i_j}{m_j^{(k)}} B_{I,M^{(k)}}(\mathbf{x}). \tag{13}$$

Substituting (13) into (10) gives a more useful representation for the $\mathbf{F}_k$:

$$\mathbf{F}_k(\mathbf{x}) = \sum_{I}^{M^{(k)}} \mathbf{v}_I^{(k)} B_{I,M^{(k)}}(\mathbf{x}) \tag{14}$$

where

$$\mathbf{v}_I^{(k)} = (\frac{i_1}{m_1^{(k)}}, \frac{i_2}{m_2^{(k)}}, \ldots, \frac{i_n}{m_n^{(k)}}, w_I^{(k)}). \tag{15}$$

The $\mathbf{v}_I^{(k)}$ are called the *control points* of $\mathbf{F}_k$. In figure 1, we show the circle $x^2 + y^2 - 1 = 0$, along with the control polyhedron of its graph.

Although at first glance this seems like an unnecessarily complicated way of posing the problem, using the parametric hypersurfaces $\mathbf{F}_k$ instead of the real-valued f_k permits use of the powerful *convex-hull property* of the multivariate Bernstein basis, which we will investigate below. As we shall see, the two techniques make use of this property in different ways, resulting in different convergence characteristics.

5 The Projected-Polyhedron Intersection Algorithm

In this section, we introduce the first of our iterative root-finding algorithms for n-dimensional problems. Although not as effective as the second technique, it is easy to visualize and simple in that it only requires two straightforward algorithms in order to implement it: one for subdividing multivariate polynomials in Bernstein form, and one for finding the convex hull of a two-dimensional set of points. This algorithm was conceived as an extension of earlier adaptive subdivision algorithms for intersecting two planar algebraic curves (see section 2).

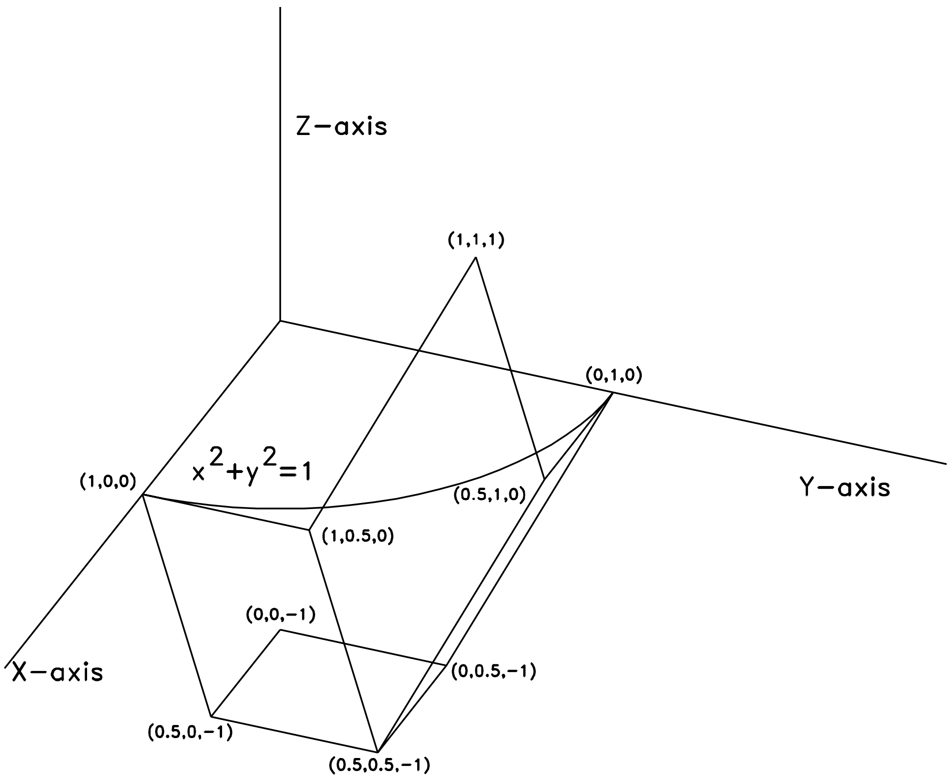

Figure 1: The control polyhedron of $(x, y, x^2 + y^2 - 1)$

5.1 Motivation for the Algorithm

Having rephrased the problem in the form of (11), we can now use properties of the Bernstein basis in order to eliminate parts of the domain $[0,1]^n$ which do not contain any roots. The two properties we will use are familiar ones which are immediate upon inspection of equation (1). The first is *positivity*:

$$b_{i,m}(u) \geq 0 \text{ for } u \in [0,1]. \tag{16}$$

The second is that the Bernstein polynomials form a *partition of unity* [9]:

$$\sum_{i=0}^{m} b_{i,m}(u) = 1. \tag{17}$$

Since the Bernstein multinomials are merely products of Bernstein polynomials, these two properties lead immediately to corresponding properties of Bernstein multinomials:

$$B_{I,M}(\mathbf{u}) \geq 0 \text{ for } \mathbf{u} \in [0,1]^n \tag{18}$$

and

$$\sum_{I}^{M^{(k)}} B_{I,M}(\mathbf{u}) = 1. \tag{19}$$

These two equations in turn lead to the important *convex hull property* of the multivariate Bernstein basis:

Theorem 5.1 *Let $V^{(k)}$ be the set of control points of $\mathbf{F}_k$ (that is, the $\mathbf{v}_I^{(k)}$ of (15)). Let $C(V^{(k)})$ be the* convex hull *of definition 3.6. Let $x \in [0,1]^n$. Then for each k between 1 and n inclusive,*

$$\mathbf{F}_k(\mathbf{x}) \in C(V^{(k)}). \tag{20}$$

Proof: Recall from (15) that $\mathbf{F}_k(\mathbf{x})$ is a linear combination of the control points $\mathbf{v}_I^{(k)}$, with coefficients being $B_{I,M^{(k)}}(\mathbf{x})$. By (18) and (19), these coefficients are positive and sum to 1, and therefore satisfy the requirements of the c_i in (5). Thus, $\mathbf{F}_k(\mathbf{x})$ is an element of the convex hull $C(V^{(k)})$.□

Because every point on the hypersurface $\mathbf{F}_k$ lies within the convex hull of its control points, equation (11) immediately implies the following theorem:

Theorem 5.2 *Let $\mathbf{x} \in [0,1]^n$ be a point satisfying (8), i.e. a root common to all of the f_k. Let X_{n+1} be the set of points of $\mathbf{R}^{n+1}$ which lie on the hyperplane $x_{n+1} = 0$. Then*

$$(\mathbf{x}, 0) \in \bigcap_{k=1}^{n} C(V^{(k)}) \cap X_{n+1} \tag{21}$$

Proof: We are given that $\mathbf{x}$ satisfies (8). Therefore, for each k between 1 and n, $\mathbf{x}$ satisfies $\mathbf{F}_k(\mathbf{x}) = (\mathbf{x}, 0)$. By theorem 5.1, the point $(\mathbf{x}, 0) \in C(V^{(k)})$, and since any point of the form $(\mathbf{x}, 0)$ is automatically an element of X_{n+1}, (21) follows.□

An simple but important corollary is worth stating:

Corollary 5.3 (Contrapositive) *Suppose $(\mathbf{x}, 0)$ does not satisfy (21). Then $\mathbf{x}$ is not a root common to all of the f_k.*

With the help of this corollary, we can devise the following general root-finding approach.

1. Let $B_0 = [0,1]^n$. This signifies that initially our region of search is the box $[0,1]^n$. As the algorithm proceeds, B_0 will keep track of which sub-box of $[0,1]^n$ we are considering.
2. Form the convex hulls of all the $\mathbf{F}_k$ and intersect them with one another and with the hyperplane $x_{n+1} = 0$. Call the intersection set A. If A is sufficiently small, stop.
3. Because A is the result of an intersection with the hyperplane $x_{n+1} = 0$, it can be expressed in the form $A = A' \times \{0\}$ for some set $A' \in [0,1]^n$. By corollary 5.3, any roots in $[0,1]^n$ must lie inside A'. We will call an element of A' a *feasible point.*
4. Find a box $B = [a_1, b_1] \times [a_2, b_2], \times \ldots \times [a_n, b_n] \in [0,1]^n$ which encloses A'. If this box B is not significantly smaller than $[0,1]^n$, split B into some number of equally sized, smaller boxes. Continue with the next step once for each one of these sub-boxes.
5. For each k, define a new function f'_k by
$$f'_k(\mathbf{x}) = f_k(a_1 + (b_1 - a_1)x_1, a_2 + (b_2 - a_2)x_2, \ldots, a_n + (b_n - a_n)x_n). \tag{22}$$
This new function may be computed by applying De Casteljau subdivision according to the box B on f_k. (See appendix B for an algorithm which implements this procedure.) Notice that f'_k maps $[0,1]^n$ to the same set that f_k maps B to.
6. Update B_0. If $B_0 = [a_{0,1}, b_{0,1}] \times [a_{0,2}, b_{0,2}] \times \ldots \times [a_{0,n}, b_{0,n}]$, then the new $a_{0,k}$ and $b_{0,k}$ are given by $a_{0,k} \leftarrow a_{0,k} + (b_{0,k} - a_{0,k})a_k$ and $b_{0,k} \leftarrow a_{0,k} + (b_{0,k} - a_{0,k})b_k$, where k ranges from 1 to n.
7. Construct the control points of the graphs $\mathbf{F}_k$ from the f_k, and go back to step 2.

Although the above procedure is sound, we need to resolve two problems before implementation. First, forming the convex hull of a set of m points in n dimensions is generally an $\mathcal{O}(m^{[\frac{n}{2}]+1})$ procedure [25]. And second, even when these convex hulls are generated, the time-consuming process of intersecting them remains, along with a large space requirement in which to store the result.

Fortunately, it turns out that it is possible to obtain a good bounding box B *without actually executing step 2.* The two algorithms we introduce in this paper take full advantage of this fact, although they use different methods to generate B. Let us now examine the way that we generate the box B in the Projected-Polyhedron algorithm.

5.2 Generating the Bounding Box

We saw that the major difficulty of our computational procedure was generating and intersecting convex hulls from a set of points. The Projected-Polyhedron method generates a bounding-box by considering n two-dimensional problems on each iteration (one for each dimension of $[0,1]^n$) instead of one $n+1$-dimensional problem. The philosophy behind this approach is to generate each of the n sides of the box independently by *projecting* control points onto different planes.

Suppose we want to find $[a_j, b_j]$, the jth side of the bounding box B. Define a function $\boldsymbol{\pi}_j : \mathbf{R}^{n+1} \to \mathbf{R}^2$ by the equation

$$\boldsymbol{\pi}_j(\mathbf{x}) = (\pi_j(\mathbf{x}), \pi_{n+1}(\mathbf{x})) \tag{23}$$

where π_j and π_{n+1} are the projection maps of definition 3.5. Using $\boldsymbol{\pi}_j$, we will map the control points of each of the $\mathbf{F}_k$ into a Cartesian plane (which for the sake of simplicity we will call the (x,y) plane). We will then generate the (two-dimensional) convex hulls of these projected points and intersect them with one another and with the x-axis. Let us call the set resulting from this intersection B_j. Since B_j is a subset of the x-axis, it can be expressed as $B'_j \times \{0\}$ for some $B'_j \in [0,1]$. As the following theorem demonstrates, the set of feasible points A' is a subset of the box $B = B'_1 \times B'_2 \times \ldots \times B'_n$:

Theorem 5.4 *Let A' be the set of feasible points, as before. Let $\mathbf{v}_{j,I}^{(k)} = \boldsymbol{\pi}_j(\mathbf{v}_I^{(k)})$ for some specific j and k, and for each I bounded by $M^{(k)}$. Let $V_j^{(k)}$ be the set of these two-dimensional points $\mathbf{v}_{j,I}^{(k)}$, and let $C(V_j^{(k)})$ be its convex hull. Let X_2 be the x-axis (the set of points in the (x,y)-plane satisfying $y=0$). Finally, let*

$$B_j = \bigcap_{k=0}^{n} C(V_j^{(k)}) \cap X_2. \tag{24}$$

Express B_j in the form $B'_j \times \{0\}$. Then if $B = B'_1 \times B'_2 \times \ldots \times B'_n$, $A' \subset B$.

Proof: Let $\mathbf{x} \in A'$. Then for some choice of coefficients $\{c_I^{(k)}\}$ such that each $c_I^{(k)} \geq 0$ and $\sum_I^{M^{(k)}} c_I^{(k)} = 1$, we must have

$$(\mathbf{x}, 0) = \sum_I^{M^{(k)}} c_I^{(k)} \mathbf{v}_I^{(k)} \tag{25}$$

for all k between 1 and n. Taking $\boldsymbol{\pi}_j$ of both sides, we obtain

$$(x_j, 0) = \sum_I^{M^{(k)}} c_I^{(k)} \mathbf{v}_{j,I}^{(k)}. \tag{26}$$

Therefore $(x_j, 0) \in C(V_j^{(k)})$ for all k. Since also $(x_j,0) \in X_2$, $(x_j, 0) \in B_j$ by (24), and thus $x_j \in B'_j$. Because this must hold for all j, we conclude that $\mathbf{x} \in B$. Since $\mathbf{x}$ was arbitrary, $A' \subset B$.□

Keep in mind that this inclusion is in general proper; that is, usually $A' \neq B$. In fact, A' tends to be *much* smaller than B in more than one dimension, resulting in a lower order of convergence than a tighter box around A' would achieve. Nevertheless, because generating the box is fairly simple, it is a reasonably good technique for moderate-sized input.

Generation of the box is essentially performed according to the theorem. Suppose we are considering the jth direction. After we generate the $\mathbf{v}_{j,I}^{(k)}$ for each k, we construct their convex hulls with a simple algorithm related to Graham's scan [6]. We then intersect these convex polygons with the interval $[0,1]$, which can be accomplished simply by marching along the vertices of each polygon and recording any intersections between their joining line segments and the x-axis as we go along. Note that since these polygons are convex, the intersection will always be either empty, a point, or a closed interval on the x-axis.

Once we have obtained these intervals (escaping from the procedure if any of them are empty) we simply intersect them. If the intersection B'_j is nonempty, it can therefore be expressed as an interval $[a_j, b_j]$.

Performing this procedure for each j will give us a new B which is either empty or equal to $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$.

Figure 2 shows the effect of projection in the case of an intersecting circle and ellipse. Notice that the projection of their convex hulls cover the entire interval $[0, 1]$ on the x-axis, giving us no useful information; however, parts of the y-axis may be eliminated immediately, telling us that there is no root of the system whose y-coordinate is less than 0.1875 or greater than 0.625. Section 7.1.1 contains a more detailed analysis of this particular problem.

5.3 The Projected-Polyhedron Algorithm

We now give full pseudocode for PROJECTED-POLYHEDRON, saving the analysis of why it works and how fast it works for the next section.

The routine takes as input the array of functions f (which represent the f_k) expressed in the multivariate Bernstein basis, and n, the number of functions in the array (and the dimension of the solution space). The arrays a and b are n-dimensional real arrays designed to represent the size of the current bounding box. On entry, the region of search is presumed to be $[0, 1]^n$. On exit, the routine returns a list of all the roots in $[0, 1]^n$ which are distinct within a given user tolerance.

```
Projected-Polyhedron (f, n)
a ← Array (n)
b ← Array (n)
for i ← 1 to n
    a[i] ← 0
    b[i] ← 1
return PP-Recur (a, b, f, n, 1)
```

```
PP-Recur (a, b, f, n, j0)
if Small-Enough (a, b)
    return Midpoint (a, b)
for j ← j0 to n
    a0 ← 0
    b0 ← 1
    for k ← 1 to n
        ajk, bjk ← Hull-Check (j, k, f, n)
        if ajk = Nil
            return No-Point
        else
            ajk ← a[j] + (b[j] − a[j])ajk
            bjk ← a[j] + (b[j] − a[j])bjk
        a0, b0 ← Intersect-Interval (a0, b0, ajk, bjk)
        if a0 = Nil
            return No-Point
    if b0 − a0 > (b[j] − a[j]) · Crit
        a′ ← a
        b′ ← b
        f′ ← f
        Subdivide (a, b, f, j, n, a0, (a0 + b0)/2)
        Subdivide (a′, b′, f′, j, n, (a0 + b0)/2, b0)
        return Concat
            (PP-Recur (a, b, f, n, (j + 1) mod n),
             PP-Recur (a′, b′, f′, n, (j + 1) mod n))
    else
        Subdivide (a, b, f, j, n, a0, b0)
return PP-Recur (a, b, f, n, 1)
```

Instead of pseudo-code, we give brief summaries of the routines called by Projected-Polyhedron and PP-Recur in order of appearance.

- Array(n) returns an array of n elements.
- Small-Enough(a, b) checks to see if the box $[a_0, b_0] \times [a_1, b_1] \times \ldots \times [a_n, b_n]$ is small enough to satisfy a given user tolerance. If so, True is returned, otherwise False.
- Midpoint(a, b) returns a list containing one element, the point $(\frac{a_0+b_0}{2}, \frac{a_1+b_1}{2}, \ldots, \frac{a_n+b_n}{2})$.
- Hull-Check(j, k, f, n) is the heart of the algorithm. Given j and k, it forms the set of projected control points $V_j^{(k)}$. It then computes the convex hull $C(V_j^{(k)})$ (using, for example, a variant of Graham's scan [6]) and intersects it with the horizontal axis by traveling along the vertices in a counterclockwise direction. If the intersection is nonempty, it returns the endpoints ajk and bjk of the intersection interval. If the intersection is empty, it makes ajk equal to the value Nil.

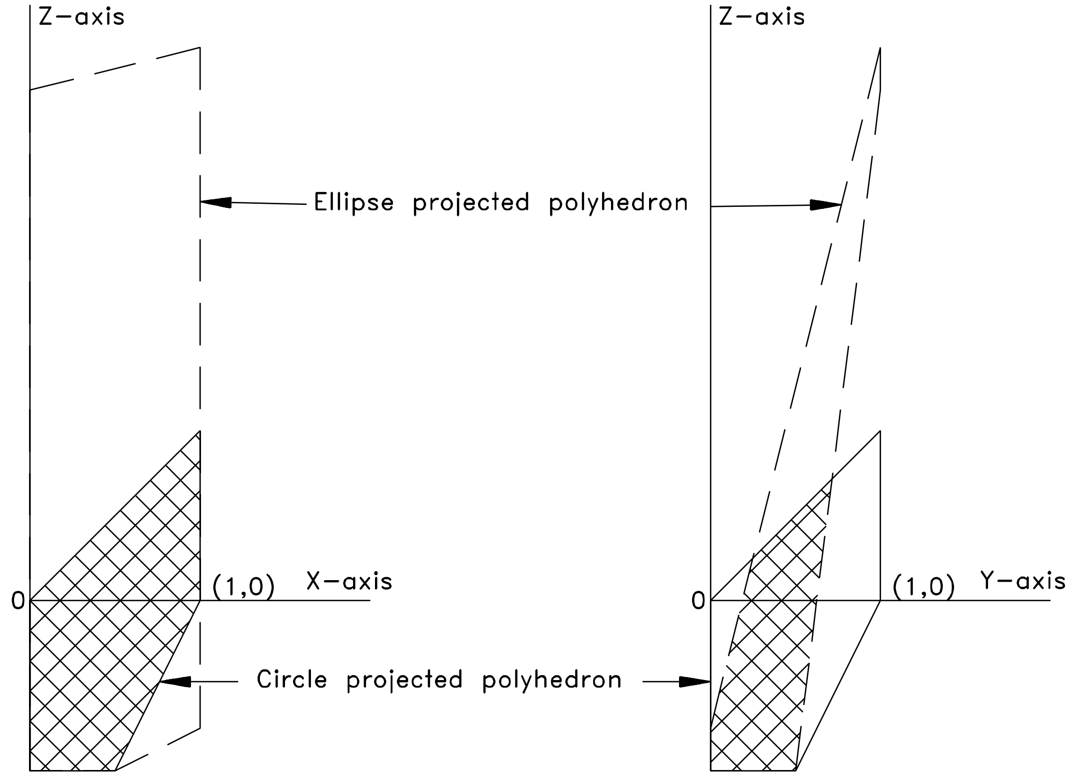

Figure 2: Effect of projecting the polyhedra of $(x, y, x^2 + y^2 - 1)$ and $(x, y, \frac{x^2}{4} + 4y^2 - 1)$

- Intersect-Interval $(a0, b0, ajk, bjk)$ returns the endpoints of the intersection of the two intervals $[a0, b0]$ and $[ajk, bjk]$.
- Subdivide $(a, b, f, j, n, a0, b0)$ does several things. First it changes $a[j]$ to the its new value $a0$ and $b[j]$ to its new value $b0$. Then it subdivides each of the f_k in the jth direction so that f_k maps $[0,1]^n$ to the portion of the function that at the beginning of execution was mapped to $[a_0, b_0] \times [a_1, b_1] \times \ldots \times [a_n, b_n]$.
- Concat $(l1, l2)$ returns the concatenation of two lists of points.

Now let us look at the pseudocode more closely, starting with the function Projected-Polyhedron. In lines 1–5, we construct the two n-dimensional arrays a and b and initialize them to all zeros and all ones respectively. This signifies that initially, our region of search is the n-dimensional box $[0,1]^n$. In line 6, we call the recursive driver PP-Recur, which we will now analyze.

In lines 1–2, the function checks to see if the input box defined by the arrays a and b is sufficiently small. If we determine that it is sufficiently small, we assume that there is one root inside and return a list containing the midpoint of the box as an approximate root. Otherwise we go on to line 3, which begins the main loop, where the index j signifies the coordinate direction currently under consideration. First we set $[a0, b0]$ equal to $[0,1]$, the maximum possible size of a side of the bounding box. Then we begin a second loop; k signifies which of the f_k is currently under consideration. At line 7, we call the function Hull-Check described above. In lines 8–9, we check to see if the intersection between the convex hull and the horizontal axis is empty; if so, we return the value No-Point to signify that there is no root within the input box. Otherwise, we execute lines 10–12, which perform scaling on the interval returned. This scaling is necessary because Hull-Check, like the preceding discussion, assumes that the input box is $[0,1]^n$. In fact, the input box is the one determined by a and b.

In line 13, we intersect $[a0, b0]$ with the interval $[ajk, bjk]$ so that at the end of the n steps, $[a0, b0]$ will be equal to the intersection of all the intervals obtained in step 12. If at any time INTERSECT-INTERVAL makes $a0$ equal to NIL, then we know that the intersection is empty, and accordingly return NO-POINT in line 15.

Otherwise, in line 16 we check the size of the interval $[a0, b0]$ compared to the interval $[a[j], b[j]]$. If the new interval is greater than some constant fraction of the original interval, we decide that the iteration is proceeding too slowly (probably because there is more than one root in the box in question) and perform binary subdivision. The value CRIT here is some fractional value between 0.5 and 1; in accordance with [31], we have mostly used the value 0.8, as it appears to give fairly good results.

Binary subdivision is performed in lines 17–24 if necessary. We start by making copies of a, b, and f. At lines 20–21, we subdivide all of the f_k in the jth direction straight down the middle of the jth component of the domain; f' becomes an array of the right halves of f, and f becomes an array of the left halves. In lines 22–24, we recurse once for each half. The function CONCAT is called to concatenate the lists of roots returned by each half. Notice that the final argument passed to PP-RECUR is the value $j+1$. This argument is passed so that the next level of recursion will begin analyzing in the $j+1$st direction, where it left off, instead of starting over at $j = 1$.

If binary subdivision is not necessary, we simply subdivide in the jth direction so that the bounding box changes its jth dimension from $[a[j], b[j]]$ to the smaller interval $[a0, b0]$. Then we proceed with the next coordinate direction by incrementing j. If $j = n$, then we perform another level of recursion with a new, smaller box.

5.4 Analysis

Up to this point, we have not attempted to analyze the performance of the Projected-Polyhedron algorithm. In this section, we will first analyze the working of the algorithm by introducing a *bounding box tree.* Next, we will briefly analyze the time complexity of the algorithm. Finally, by analyzing performance on a local scale, we will show that the algorithm achieves quadratic convergence for $n = 1$ and at best linear convergence for $n > 1$.

5.4.1 The Bounding-Box Tree

Let us first prove that the algorithm actually completes within a finite number of steps. In order to do this, we need to first define a few terms relating to the algorithm:

Definition 5.5

1. *A* step s *is one execution of the j loop in lines 3–26, not counting any recursive calls to* PP-RECUR. *Clearly, there are a maximum of n steps during any particular call of* PP-RECUR.
2. *The* bounding box B *of step s is the box defined by the n-dimensional arrays a and b when the step begins.*
3. *A* child *of a bounding box B is a bounding box which results from subdivision of B in line 26 or one of the two boxes resulting from subdivision in lines 20–21.* Parent *denotes the reverse relationship. Note that there are no more than two children of any given box.*

With the parent-child relationship defined among bounding boxes, we can construct a *bounding-box tree* where the nodes are bounding boxes and the links represent this relationship.

Lemma 5.6 *Let B_1 be a given node of the bounding-box tree. Suppose B_2 is another node of the tree which is nth in descent from B_1. (First in descent is a child, second is a grandchild, and so on.) Then each side of B_2 is no larger than the corresponding side of B_1 times* CRIT^n.

Proof: By the definition of a step, between B_1 and B_2 there are n executions of the j loop in lines 3–26. But j is incremented each time modulo n (even if recursion is performed in lines 23–24, we continue in the next level of recursion at $(j+1) \bmod n$), and so each of the n different coordinate directions is considered in turn. Now on each step, either binary subdivision is performed in lines 20–21 or else subdivison is performed in line 26. In the first case, each child box is exactly half as large in the jth direction as its parent; in the second, the child box is no more than CRIT (a value between 0.5 and 1) times as large in the jth direction as its parent. In either case, the other sides of the box remain undisturbed. Therefore, at each step, the child box is smaller on the jth side by at least a factor of CRIT. Since each step reduces a different side and there are n steps, we conclude that each side of B_2 is no larger than the corresponding side of B_1 times CRIT^n. □

Corollary 5.7 *Suppose we are given some number $\varepsilon > 0$. Then there exists a number N such that after N or fewer steps, we generate a bounding box tree whose leaves (terminal nodes) are either bounding boxes with every side smaller than ε or bounding boxes which have been determined to contain no root.*

Proof: Since CRIT < 1, there exists some integer $p > 0$ such that $\text{CRIT}^p < \varepsilon$. Now each node must have at least one child unless either

1. The box is sufficiently small (in this case, each side must be less than ε), or
2. NO-POINT is returned in line 9 or line 15 because no root exists in the box.

Therefore, a tree of depth np will have as leaves boxes with sides less than ε or boxes which contain no root. This tree has at most 2^{np} nodes, so setting $N = 2^{np}$ completes the proof.□

By itself, Corollary 5.7 is not particularly reassuring. We would like all of the boxes of the first type (that is, boxes under tolerance) to contain one and only one root, but the corollary gives no assurance of this by itself. However, as we shall see, for small n the local convergence properties of the algorithm minimize this problem; in most cases, for reasonably small ε, all of the boxes under tolerance will contain one and only one root. Of course, the actual number of steps needed to complete execution will usually be far smaller than 2^{np}; this value is merely an upper bound useful for showing that the algorithm terminates.

5.4.2 Complexity Analysis

It is difficult *a priori* to predict how many steps will actually be needed to complete the algorithm. However, we can analyze the amount of time required to execute each *step* of the algorithm individually (see definition 5.5). A more useful measurement is the amount of time needed to execute n steps, since if no binary subdivision is required, one call to PP-RECUR takes n steps. In the analysis, we will assume that each equation is degree m in each variable, or of total degree mn; although it is easy to extend the analysis to the general case that all the degrees are different, the resulting formulas are quite long.

The generation of a convex hull of $(m+1)^n$ two-dimensional points takes at most $\mathcal{O}((m+1)^n \log (m+1)^n)$ time [6]. However, by exploiting the special structure of these points, we can reduce this cost. There are several points with identical abscissae; however, at most two of them can belong to the boundary of the convex hull (specifically, the points with the minimum and maximum ordinates). Therefore, we can discard all other points from consideration, leaving us with $2(m+1)$ points (a minimum and maximum point for each abscissa). These points may now be traversed in counterclockwise order as in Graham's scan [6]. The entire execution time of this procedure is $\mathcal{O}((m+1)^n)$. The time needed to traverse this convex hull in order to intersect it with the x-axis is no greater, so we conclude that the total execution time of HULL-CHECK in line 7 of Projected-Polyhedron is $\mathcal{O}(m^n)$. Lines 8–15 do not take enough time to affect the asymptotic total, so we omit them from the analysis. Because the loop in lines 7–15 is executed n times, the total cost of these lines becomes $\mathcal{O}(nm^n)$. Let us assume that binary subdivision is not necessary (i.e. the condition in line 16 is false). A glance at the pseudocode in appendix B will indicate that to subdivide one of the f_k in one direction takes $\mathcal{O}(m^{n+1})$ time; since there are n f_k which are subdivided in SUBDIVIDE, the total time needed to execute line 26 is $\mathcal{O}(nm^{n+1})$. Note that if binary subdivision is needed instead then the time is simply twice as long, which does not affect the asymptotic time.

Thus the total asymptotic time of one step (i.e. one execution of the j loop in lines 3–26) is $\mathcal{O}(nm^n + nm^{n+1})$, which means that the time needed to run n steps is $\mathcal{O}(n^2 m^{n+1})$. Although at first it may seem unpleasant that the algorithm is exponential in n, keep in mind that the input itself (the control points of the polynomials) is itself exponential in n, so merely printing out the control points would take exponential time. Furthermore, for most geometric modeling applications, it is quite rare to see an n higher than 6. It is often more helpful to consider that for a given number of dimensions, the algorithm is polynomial in m.

This analysis is incomplete because the number of steps needed may be quite different for systems with the same n and m. However, the local convergence properties can give us a better idea of how much time is required. In order to analyze these local convergence properties, we will first examine the relationship between the level of subdivision of a multivariate polynomial and the closeness of its control points.

5.4.3 Establishing Bounds

The purpose of this subsection is to prove the following theorem:

Theorem 5.8 *Let* $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ *and* $\mathbf{b} = (b_1, b_2, \ldots, b_n)$. *Let* $\boldsymbol{\varepsilon} = (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n) = (b_1 - a_1, b_2 - a_2, \ldots, b_n - a_n)$. *Suppose* $f : [0,1]^n \to \mathbf{R}$ *is the function which results from subdividing another function* $f_0 : [0,1]^n \to \mathbf{R}$ *over the box* $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$; *that is,*

$$f(x_1, x_2, \ldots, x_n) = f_0(a_1 + x_1\varepsilon_1, a_2 + x_2\varepsilon_2, \ldots, a_n + x_n\varepsilon_n). \tag{27}$$

Suppose further that for a given bounding multi-index $M = (m_1, m_2, \ldots, m_n)$, $f(\mathbf{x}) = \sum_I^M p_I B_{I,M}(\mathbf{x})$. *Then there exists a positive constant* κ *independent of* $\boldsymbol{\varepsilon}$ *such that for each* I *bounded by* M, *there exists a point* $\mathbf{x} \in [0,1]^n$ *for which* $|f(\mathbf{x}) - p_I| < \kappa|\boldsymbol{\varepsilon}|^2$.

Informally, this theorem simply says that if the sides of our bounding box are on the order of $|\boldsymbol{\varepsilon}|$, then the distance from f to any of its control points is on the order of $|\boldsymbol{\varepsilon}|^2$.

In order to prove theorem 5.8, we will break the problem up into several steps. First, we take a Taylor expansion of f around the point $\mathbf{x} = \mathbf{0}$ in order to generate a linear approximation $g : [0,1]^n \to \mathbf{R}$:

$$g(\mathbf{x}) = f(\mathbf{0}) + \sum_{j=1}^{n} x_j \frac{\partial f}{\partial x_j}(\mathbf{0}) \tag{28}$$

Since g is linear in each coordinate direction, it can be written as the sum of Bernstein multinomials of degree 1 in each variable. Suppose we now apply *degree elevation* [9] in each of the n directions, so that coordinate direction j has degree m_j instead of 1. Then we can rewrite g as

$$g(\mathbf{x}) = \sum_{I}^{M} q_I B_{I,M}(\mathbf{x}) \tag{29}$$

An important property of g is that because g is linear, all of its control points q_I lie on the image set of g. See, for example, [9] for a more detailed discussion and proof of this property.

Another important characteristic of g is that because it is a first-order approximation to f, the difference $h \equiv f - g$ is a second-order quantity. That is, there exists some positive constant κ_0 independent of $\boldsymbol{\varepsilon}$ such that for all $\mathbf{x} \in [0,1]^n$,

$$|h(\mathbf{x})| \leq \kappa_0 |\boldsymbol{\varepsilon}|^2 \tag{30}$$

Since $h = f - g$, we can also write

$$h(\mathbf{x}) = \sum_{I}^{M} \delta_I B_{I,M}(\mathbf{x}) \tag{31}$$

where $\delta_I \equiv p_I - q_I$. We would like now to establish bounds on the δ_I, which we do by first proving bounds on the derivatives of h evaluated at $\mathbf{0}$:

Lemma 5.9 *Let $I = (i_1, i_2, \ldots, i_n)$, and let the notation $\frac{\partial^I}{\partial \mathbf{x}^I}$ be equivalent to $\frac{\partial^{i_1+i_2+\ldots+i_n}}{\partial x_1^{i_1} \partial x_2^{i_2} \ldots \partial x_n^{i_n}}$. Let h be as above. Then there exists a positive constant κ_1 independent of ε such that $|\frac{\partial^I h}{\partial \mathbf{x}^I}(\mathbf{0})| \leq \kappa_1 |\varepsilon|^2$ for all I bounded by M.*

Proof: There are three cases we need to consider:

Case I: If $I = (0, 0, \ldots, 0)$, then $\frac{\partial^I h}{\partial \mathbf{x}^I} = h$. Since $g(\mathbf{0}) = f(\mathbf{0})$, $h(\mathbf{0}) = 0 < \kappa_1 |\varepsilon|^2$ for any $\kappa_1 > 0$.

Case II: If $\frac{\partial^I h}{\partial \mathbf{x}^I} = \frac{\partial h}{\partial x_j}$ for some j (that is, $i_j = 1$ for some j and the other $n-1$ elements of I are 0), then $\frac{\partial^I h}{\partial \mathbf{x}^I}(\mathbf{0}) = 0 < \kappa_1 |\varepsilon|^2$ because $\frac{\partial f}{\partial x_j}(\mathbf{0}) = \frac{\partial g}{\partial x_j}(\mathbf{0})$.

Case III: For any I bounded by M which does not satisfy either case above, $\frac{\partial^I h}{\partial \mathbf{x}^I} = \frac{\partial^I f}{\partial \mathbf{x}^I}$ because derivatives of g of order greater than 1 are identically 0. Now recall that f is a function which results from subdividing another function f_0 over the box $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$. Therefore, we can make use of equation (27) and the chain rule to obtain

$$\frac{\partial^I f}{\partial \mathbf{x}^I}(\mathbf{0}) = \frac{\partial^I f_0}{\partial \mathbf{x}^I}(\mathbf{a}) \prod_{j=1}^{n} \varepsilon_j^{i_j}. \tag{32}$$

Because I does not satisfy either of the first two cases, we must have $i_1 + i_2 + \ldots + i_n \geq 2$. Therefore,

$$\prod_{j=1}^{n} \varepsilon_j^{i_j} \leq |\varepsilon|^2 \tag{33}$$

Furthermore, since f_0 is polynomial, all of its derivatives are bounded over $[0,1]^n$. Thus there exists some positive constant κ_1 such that for all $\mathbf{x} \in [0,1]^n$ and all I bounded by M,

$$|\frac{\partial^I f_0}{\partial \mathbf{x}^I}(\mathbf{x})| \leq \kappa_1. \tag{34}$$

Substituting (34) and (33) into (32) finally gives

$$|\frac{\partial^I h}{\partial \mathbf{x}^I}(\mathbf{0})| \leq \kappa_1 |\varepsilon|^2. \tag{35}$$

Since these three cases cover all permissible I, we are done.□

The next step is to show that each of the δ_I can be expressed as a linear combination of derivatives of h evaluated at $\mathbf{0}$. In order to do this, we recall the following result from [10]:

$$x_j^k = \sum_{i=k}^{n} \frac{\binom{i}{k}}{\binom{n}{k}} b_{i,n}(x_j). \tag{36}$$

Now h can be as easily expressed in the power multinomial basis as the Bernstein multinomial basis; that is, there exist coefficients d_I such that

$$\sum_I^M \delta_I B_{I,M}(\mathbf{x}) = \sum_I^M d_I \prod_{j=1}^{n} x_j^{i_j}. \tag{37}$$

Substituting (36) for each x_j, we obtain

$$h(\mathbf{x}) = \sum_I^M d_I \prod_{j=1}^{n} [\sum_{i=i_j}^{m_j} \frac{\binom{i}{i_j}}{\binom{m_j}{i_j}} b_{i,m_j}(x_j)]. \tag{38}$$

If we define $\binom{n}{k} \equiv 0$ for $n < k$, (38) becomes

$$h(\mathbf{x}) = \sum_{I}^{M} d_I \prod_{j=1}^{n} [\sum_{i=0}^{m_j} \frac{\binom{i}{i_j}}{\binom{m_j}{i_j}} b_{i,m_j}(x_j)]. \tag{39}$$

Now we can easily equate like terms to obtain the following relationship between δ_I and d_I:

$$\delta_I = \sum_{J}^{M} \frac{\binom{j_1}{i_1}\binom{j_2}{i_2}\dots\binom{j_n}{i_n}}{\binom{m_1}{i_1}\binom{m_2}{i_2}\dots\binom{m_n}{i_n}} d_J. \tag{40}$$

This equation may not appear extremely relevant at first sight, but it becomes more useful when we recall the simple relationship between the d_J and the derivatives of h at $\mathbf{0}$:

$$d_J = \frac{1}{j_1!j_2!\dots j_n!} \frac{\partial^J h}{\partial \mathbf{x}^J}(\mathbf{0}) \tag{41}$$

This relationship can be readily verified by differentiating the right-hand side of (37), but it is a familiar result of multivariable calculus. By plugging (41) into (40) we obtain

$$\delta_I = \sum_{J}^{M} \frac{\binom{j_1}{i_1}\binom{j_2}{i_2}\dots\binom{j_n}{i_n}}{\binom{m_1}{i_1}\binom{m_2}{i_2}\dots\binom{m_n}{i_n}} \frac{1}{j_1!j_2!\dots j_n!} \frac{\partial^J h}{\partial \mathbf{x}^J}(\mathbf{0}). \tag{42}$$

With this relationship established, we can prove the following lemma:

Lemma 5.10 *Let δ_I be as before. Then there exists some positive constant κ_2 independent of ε such that for all I bounded by M, $|\delta_I| \leq \kappa_2 |\varepsilon|^2$.*

Proof: Consider the coefficient of $\frac{\partial^J h}{\partial \mathbf{x}^J}(\mathbf{0})$ in equation (42). Since J is bounded by M, $m_k \geq j_k$ for $k \in \{1, 2, \dots, n\}$, and so $\binom{m_k}{i_k} \geq \binom{j_k}{i_k}$. Therefore, each coefficient is a positive number less than or equal to 1, giving

$$|\delta_I| \leq \sum_{J}^{M} |\frac{\partial^J h}{\partial \mathbf{x}^J}(\mathbf{0})|. \tag{43}$$

Using the result of lemma 5.9, we obtain

$$|\delta_I| \leq m_1 m_2 \dots m_n \kappa_1 |\varepsilon|^2 \tag{44}$$

Putting $\kappa_2 \equiv m_1 m_2 \dots m_n \kappa_1$ completes the proof.□

Now we are ready to put all the pieces together and prove theorem 5.8.

Proof of Theorem: Pick a multi-index I bounded by M. Now, q_I lies on the image set of g, i.e. there exists a point $\mathbf{x} \in [0,1]^n$ such that

$$g(\mathbf{x}) = q_I \tag{45}$$

From lemma 5.10 we know that

$$|p_I - q_I| = |g(\mathbf{x}) - p_I| \leq \kappa_2 |\varepsilon|^2 \tag{46}$$

But equation (30) gives us a bound on the quantity $|f - g|$. Applying the triangle inequality, we get

$$|f(\mathbf{x}) - p_I| \leq |f(\mathbf{x}) - g(\mathbf{x})| + |g(\mathbf{x}) - p_I| \leq (\kappa_0 + \kappa_2)|\varepsilon|^2. \tag{47}$$

Setting $\kappa \equiv \kappa_0 + \kappa_2$ completes the proof.□

5.4.4 Quadratic Convergence in One Dimension

With the aid of theorem 5.8, we will now prove the following theorem, which states that PP-RECUR achieves quadratic convergence in one dimension:

Theorem 5.11 *Suppose we are given an interval* $[a, b]$ *which contains one and only one root* x_0 *of a polynomial* $f_0 : [0, 1] \to \mathbf{R}$. *Further assume that* x_0 *is a simple root. Let* $\varepsilon \equiv b - a$, *and let* f *denote* f_0 *subdivided over* $[a, b]$, *i.e.* $f(x) = f_0(a + x\varepsilon)$. *Let* ε' *denote the size of the new interval which results after executing lines 4–13 of* PP-RECUR *with* f, a, *and* b *as the governing parameters. Then there exists a positive constant* λ *independent of* ε *and a* $\delta > 0$ *such that if* $\varepsilon \leq \delta$, *then* $\varepsilon' \leq \lambda\varepsilon^2$.

Proof: Now x_0 is a simple root of f_0, so $f_0'(x_0) \neq 0$. By continuity, there exists a δ such that $f_0'(x) \neq 0$ for $x \in [x_0 - \delta, x_0 + \delta]$. By assumption, $\varepsilon \leq \delta$, so $[a, b] \subset [x_0 - \delta, x_0 + \delta]$.

Let $f(x) = \sum_{i=0}^{m} p_i b_{i,m}(x)$. Let g be the linear approximation to f at 0, i.e. $g(x) = f(0) + xf'(0) = \sum_{i=0}^{m} q_i b_{i,m}(x)$. As before, let us construct the graphs of f and g:

$$\mathbf{F}(x) = (x, f(x)), \quad \mathbf{G}(x) = (x, g(x)) \tag{48}$$

From lemma 5.10, we know that for each i, $|q_i - p_i| \leq \kappa_2\varepsilon^2$. Therefore, for each i, the control points $(\frac{i}{m}, p_i)$ of $\mathbf{F}$ and $(\frac{i}{m}, q_i)$ of $\mathbf{G}$ are separated by a vertical distance less than or equal to $\kappa_2\varepsilon^2$. Since $\mathbf{G}$ is a line, all of its control points lie on the line; thus, each control point of $\mathbf{F}$ must lie on or between the following two lines which are respectively above and below $\mathbf{G}$:

$$\mathbf{L}_1(t) = (0, f(0) + \kappa_2\varepsilon^2) + (t, tf'(0)) \tag{49}$$

$$\mathbf{L}_2(t) = (0, f(0) - \kappa_2\varepsilon^2) + (t, tf'(0)) \tag{50}$$

The region enclosed by these two lines and the vertical lines $x = 0$ and $x = 1$ is convex; therefore, the convex hull of the control points of $\mathbf{F}$ is entirely contained within this region.

The intersection of L_1 with the horizontal axis is at $t = -\frac{f(0)+\kappa_2\varepsilon^2}{f'(0)}$, and the intersection of L_2 with the horizontal axis is at $t = -\frac{f(0)-\kappa_2\varepsilon^2}{f'(0)}$. The size of the intersection interval is therefore

$$\Delta t = |\frac{2\kappa_2\varepsilon^2}{f'(0)}| = |\frac{2\kappa_2\varepsilon}{f_0'(a)}| \tag{51}$$

where we have used the fact that $f'(0) = \varepsilon f_0'(a)$. Since the convex hull of the control points of $\mathbf{F}$ is completely contained in the region bounded by L_1 and L_2, the size of the interval returned by HULL-CHECK must be less than or equal to Δt. Remember, however, that this interval must be scaled in lines 11–12 of PP-RECUR; after scaling, the size ε' of the new interval is given by

$$\varepsilon' \leq |\frac{2\kappa_2\varepsilon^2}{f_0'(a)}| \tag{52}$$

Let c equal the minimum value of $|f_0'|$ over the interval $[x_0 - \delta, x_0 + \delta]$, and let $\lambda \equiv \frac{2\kappa_2}{c}$. Then (52) implies that

$$\varepsilon' \leq \lambda\varepsilon^2. \tag{53}$$

Note that since κ_2 and c are independent of ε, so is λ, and therefore the algorithm exhibits quadratic convergence in one dimension.□

5.4.5 Convergence in Higher Dimensions

Unfortunately, the quadratic convergence of one dimension does not carry over into higher dimensions. Instead, we observe at best linear convergence which, as we shall see, is due to the use of projections. Projection of control points simplifies the problem, but we pay for the privilege with a lower order of convergence.

Theorem 5.12 *Suppose we are given a box $B = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$ which contains one and only one root $\mathbf{x}_0$ of a system of polynomials $f_{0,k} : [0,1]^n \rightarrow \mathbf{R}$ where k ranges from 1 to n. Assume further that $\mathbf{x}_0$ is a simple root. Suppose f_k represents the subdivision of $f_{0,k}$ over B. For a specific value of j between 1 and n, let $\varepsilon \equiv b_j - a_j$, and execute lines 4–13 of* PP-Recur *with this particular j. Let $\varepsilon' \equiv b0 - a0$, where $a0$ and $b0$ are the values of the $a0$ and $b0$ in line 13. Then ε' is an $\mathcal{O}(\varepsilon)$ quantity.*

Proof: For the moment, fix k. Let $f_k(\mathbf{x}) = \sum_I^{M^{(k)}} p_I^{(k)} B_{I,M}(\mathbf{x})$, and let $g_k(\mathbf{x}) = \sum_I^{M^{(k)}} q_I^{(k)} B_{I,M}(\mathbf{x})$ be the degree-elevated linear approximation to f_k. Let $\mathbf{F}_k(\mathbf{x}) = (\mathbf{x}, f_k(\mathbf{x}))$ and $\mathbf{G}_k(\mathbf{x}) = (\mathbf{x}, g_k(\mathbf{x}))$ be the graphs of f_k and g_k. Let the control points of $\mathbf{F}_k$ be $\mathbf{p}_I^{(k)}$ and the control points of $\mathbf{G}_k$ be $\mathbf{q}_I^{(k)}$.

Apply the projection function $\boldsymbol{\pi}_j = (\pi_j, \pi_{n+1})$ to the $\mathbf{p}_I^{(k)}$ and the $\mathbf{q}_I^{(k)}$ in order to generate sets of two-dimensional points $\mathbf{p}_{j,I}^{(k)}$ and $\mathbf{q}_{j,I}^{(k)}$:

$$\mathbf{p}_{j,I}^{(k)} = (\pi_j(\mathbf{p}_I^{(k)}), \pi_{n+1}(\mathbf{p}_I^{(k)})) = (\frac{j}{m_j^{(k)}}, p_I^{(k)}) \tag{54}$$

$$\mathbf{q}_{j,I}^{(k)} = (\frac{j}{m_j^{(k)}}, q_I^{(k)}). \tag{55}$$

Now, by lemma 5.10, we know that $|p_I^{(k)} - q_I^{(k)}| \leq \kappa_2 \varepsilon^2$. Therefore, the points $\mathbf{p}_{j,I}^{(k)}$ and $\mathbf{q}_{j,I}^{(k)}$ are separated by a vertical distance of not more than $\kappa_2 \varepsilon^2$.

Unfortunately, the projection of $\mathbf{G}_k$ into the (x_j, x_{n+1})-plane is not a straight line if $n > 1$; in fact, it is generally a planar region with a nonzero area. For a fixed x_j, the difference between the maximum y-coordinate and the minimum y-coordinate of this region is given by

$$\Delta y = \sum_{i=1, i \neq j}^{n} |\frac{\partial f_k}{\partial x_i}(\mathbf{0})|. \tag{56}$$

This equation is obtained by taking the minimum value of (28) over all $\mathbf{x} \in [0,1]^n$ with x_j fixed and subtracting it from the maximum value of the same quantity. Now, since this region is the projection of a hyperplanar region into two dimensions, it must be a parallelogram, bounded above and below by two parallel lines separated by Δy and on the sides by the vertical lines $x = 0$ and $x = 1$. Notice that if $n = 1$, Δy is identically 0.

Using this value of Δy, we can proceed as we did in the previous section and find two lines $\mathbf{L}_1$ and $\mathbf{L}_2$ which have slope equal to $\frac{\partial f}{\partial x_j}$ and which are separated by a distance $\Delta y + 2\kappa_2 \varepsilon^2$. Together with the vertical lines $x = 0$ and $x = 1$, these lines form a region enclosing the control points of $\mathbf{F}_k$; since this region is convex, it also encloses the convex hull of the control points.

Following the same steps as the proof of theorem 5.11, we can show that the intersection between this region and the x-axis is an interval with width

$$\Delta t = |\frac{\Delta y + 2\kappa_2 \varepsilon^2}{\frac{\partial f}{\partial x_j}(\mathbf{0})}|. \tag{57}$$

But the numerator and denominator of Δt are both $\mathcal{O}(\varepsilon)$ quantities if Δy and $\frac{\partial f}{\partial x_j}(\mathbf{0})$ are both nonzero, and so Δt is an $\mathcal{O}(1)$ quantity. Now, we still have to intersect this interval with the other $n-1$ intervals obtained using different values of k; after intersection, the final interval may be considerably smaller than Δt. But if, on the other hand, for each k the intervals are about the same size, their intersection may well be about as large or the same size as Δt. Therefore, this intersection is an $\mathcal{O}(1)$ quantity. After scaling it in lines 11–12, we finally obtain $\varepsilon' = \mathcal{O}(\varepsilon)$.□

It is possible to construct special cases for $n > 1$ where one can observe quadratic or close to quadratic convergence. However, most multivariate systems of polynomial equations demonstrate linear convergence in accordance with this theorem; typically the constant hidden inside the $\mathcal{O}(\varepsilon)$ expression increases with degree and number of variables, as one might expect. Intuitively, a higher number of variables makes the projected area "thicker" and a higher degree typically makes the intersecting hypersurfaces less like hyperplanes; both of these factors will contribute to a larger constant.

In the preceding analysis, we saw that the extra "thickness" of the intersecting convex hulls due to projection reduces the order of convergence from quadratic to linear. Since the intersection region decreases in width quadratically, we expect there to be a tighter box around the region which gives rise to a quadratically convergent algorithm. As we shall see in the next section, we can use linear programming to determine such a box; this technique gives rise to a quadratically convergent algorithm regardless of dimension, which performs exceptionally well for $n = 2$.

6 The Linear Programming Root Isolation Algorithm

In this section, we present and analyze the second of our root-finding algorithms for systems of multivariate polynomials. Since most of the derivation of this algorithm is given in section 5, we will concentrate mostly on those aspects of the Linear Programming (LP) algorithm which are different from the Projected-Polyhedron technique.

The principal difference between the two methods is the way that they generate bounding boxes for the set of feasible points A' (see figure 3). Therefore, we begin by examining the way this box is generated in the LP approach.

In the following discussion we will be working with both of the two sets A and A'. Recall that $A = A' \times \{0\}$; in other words, for every element of A', there is an element of A with the first n coordinates identical and an $n+1$st coordinate equal to 0. Technically speaking, the linear programming subroutine will be searching for minima and maxima of a linear function over the set A, but these extrema will be used to bound A'. Distinguishing between the two sets A and A' is unimportant, since they are isomorphic.

6.1 Generating the Bounding Box

We start by splitting the problem of determining $B = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$ into the $2n$ independent subproblems of computing each a_i and each b_i. If we are attempting to compute a_i, the question we ask is "if we look at the ith coordinates of all points lying in A, what is the value of the smallest?" If we are looking for b_i, the end of that statement would read "what is the value of the biggest?" More formally,

$$a_i = \min_{\mathbf{x} \in A}\{x_i\}, \quad b_i = \max_{\mathbf{x} \in A}\{x_i\} \tag{58}$$

where i ranges from 1 to n. The proof that A' lies within the box determined by the a_i and the b_i is trivial, so we omit any proof of this fact.

Now, as (58) currently stands, it would appear that we need to have the set A to work with. However, we have already stated that the intersection set A is difficult to compute explicitly. The important thing to realize is that we do not actually have to determine A in order to evaluate (58); instead we write down a set of *constraint equations* which uniquely determine A. Then we run an optimization algorithm to determine the minimum and maximum values of x_i subject to these constraints. Because all of the constraints turn out to be linear, the optimization algorithm we use is a linear programming algorithm.

It turns out that the constraint equations are quite easy to write down. In order for a point $\mathbf{x} \in \mathbf{R}^{n+1}$ to be in A, the following two conditions must hold:

1. For each k between 1 and n, $\mathbf{x} \in C(V^{(k)})$.

2. $x_{n+1} = 0$.

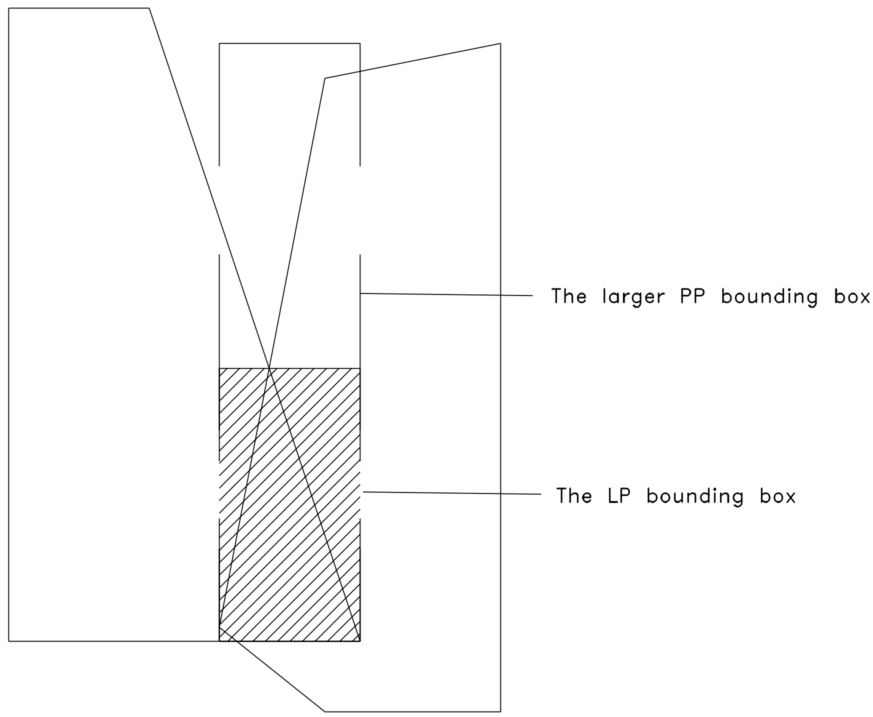

Figure 3: A comparison between a PP and an LP bounding box

By the definition of A, these conditions are also sufficient; that is, if the conditions hold for a given point $\mathbf{x}$, then $\mathbf{x} \in A$.

In order to use linear programming, we need to rewrite the constraint equations using definition 3.6. Recall that $\mathbf{x} \in C(V^{(k)})$ if and only if there exist $c_I^{(k)}$ such that

$$\mathbf{x} = \sum_{I}^{M^{(k)}} c_I^{(k)} \mathbf{v}_I^{(k)} \tag{59}$$

where the $c_I^{(k)}$ are nonnegative and sum to 1, $\mathbf{v}_I^{(k)} \in V^{(k)}$, and $\mathbf{v}_I^{(k)} \neq \mathbf{v}_J^{(k)}$ if $I \neq J$.

Using equation (15) to substitute for the $\mathbf{v}_I^{(k)}$, we can rewrite the two conditions above in a more useful form:

Theorem 6.1 *A point $\mathbf{x} \in \mathbf{R}^{n+1}$ is in A if and only if there exist nonnegative coefficients $c_I^{(k)}$ such that*

1. *for each j between 1 and n and for each k between 1 and n such that $m_j^{(k)} \neq 0$,*

$$\sum_{I}^{M^{(k)}} c_I^{(k)} \frac{i_j}{m_j^{(k)}} = x_j, \tag{60}$$

2. *for each k between 1 and n,*

$$\sum_{I}^{M^{(k)}} c_I^{(k)} w_I^{(k)} = 0, \tag{61}$$

3. and for each k between 1 and n,

$$\sum_{I}^{M^{(k)}} c_I^{(k)} = 1. \tag{62}$$

Suppose that we wish to solve (58) — that is, find the minimum value of x_i over all $\mathbf{x} \in A$ for some specific i. Then, since (60) must hold for $i = j$ and for all k, minimizing x_i is equivalent to minimizing

$$x_i = \sum_{I}^{M^{(1)}} c_I^{(1)} \frac{i_i}{m_i^{(1)}} \tag{63}$$

subject to (60), (61), and (62).

Now, linear programming problems are usually posed as follows: given a $p \times q$ matrix C of constraint function coefficients, a q-vector $\mathbf{u}$ of nonnegative unknowns, a p-vector $\mathbf{r}$ of right-hand sides, and a q-vector $\mathbf{c}$ of objective function coefficients, minimize $\mathbf{c}^T\mathbf{u}$ over all possible $\mathbf{u}$ which satisfy $C\mathbf{u} = \mathbf{r}$ [13]. With a small amount of algebra, we can rewrite our minimization problem in this form. Note that if we want to find the *maximum* of $\mathbf{c}^T\mathbf{u}$ instead, we can simply negate all of the elements of $\mathbf{c}$ and obtain a minimization problem.

The vector $\mathbf{u}$ of unknowns will simply be all of the $c_I^{(k)}$. Therefore, q, the number of elements of $\mathbf{u}$, will be given by

$$q = \sum_{k=1}^{n} (m_1^{(k)} + 1)(m_2^{(k)} + 1)\dots(m_n^{(k)} + 1) \tag{64}$$

The $2n$ constraints obtained by writing (61) and (62) for each k may be directly inserted into C. However, in order to put (60) into C for all j and k which satisfy $m_j^{(k)} \neq 0$, we must eliminate the x_j from the equations. When we have done this, the first of the three conditions of theorem 6.1 may be restated as follows:

Corollary 6.2 *For all j between 1 and n and all k between 1 and $n-1$ such that $m_j^{(k)} \neq 0$, if $\mathbf{x}$ is in A and if there exists a k' which is the smallest integer such that $k < k' \leq n$ and $m_j^{(k)} \neq 0$, then*

$$\sum_{I}^{M^{(k)}} c_I^{(k)} \frac{i_j}{m_j^{(k)}} - \sum_{I}^{M^{(k')}} c_I^{M^{(k')}} \frac{i_j}{m_j^{(k')}} = 0. \tag{65}$$

Notice that if $m_j^{(k)} \neq 0$ for all j and k, then $k' = k + 1$.

We insert these equations (there are no more than $n^2 - n$) into C directly, resulting in a total number of constraint equations $p \leq n^2 + n$.

In section 7 we will give a few examples which show how to create C, $\mathbf{y}$, and $\mathbf{c}$. But first, let's use this approach to build a recursive algorithm called LP-FIND, similar in spirit to PROJECTED-POLYHEDRON.

6.2 The LP Algorithm

As we shall see, LP-FIND looks much like PROJECTED-POLYHEDRON. However, there are two major differences between the two algorithms. First, of course, is that LP-FIND uses linear programming in order to generate a new bounding box. Second is the way LP-Find handles slowing of convergence; we will examine this in more detail below.

LP-FIND takes as input the same parameters as PROJECTED-POLYHEDRON. On exit, it returns a list of all the roots in $[0,1]^n$ which are distinct within a given user tolerance.

```
LP-FIND (f, n)
  a ← ARRAY (n)
  b ← ARRAY (n)
  for i ← 1 to n
      a[i] ← 0
      b[i] ← 1
  return LP-RECUR (a, b, f, n)
```

```
LP-RECUR (a, b, f, n)
  if SMALL-ENOUGH (a, b)
      return MIDPOINT (a, b)
  C, y, c, p, q ← CREATE-BOX (f, n)
  cneg ← ARRAY (q)
  sub ← ARRAY (n)
  for i ← 1 to q
      cneg[i] ← −c[i]
  for i ← 1 to n
      u ← LP-SOLVER (C, y, c, p, q)
      if u = NO-POINT
        return NO-POINT
      a0 ← DOT-PRODUCT (c, u)
      u ← LP-SOLVER (C, y, cneg, p, q)
      b0 ← DOT-PRODUCT (c, u)
      if (b0 − a0) > CRIT
        sub[i] = TRUE
      else
        sub[i] = FALSE
      a0 ← a[i] + (b[i] − a[i])a0
      b0 ← a[i] + (b[i] − a[i])b0
      a[i] ← a0
      b[i] ← b0
  return SPAWN (a, b, f, n, sub)
```

Here are summaries of the routines called by LP-FIND and LP-RECUR which have not already been described in the explanation of PROJECTED-POLYHEDRON:

- CREATE-BOX(f, n) computes the integers p and q and creates the $p \times q$ matrix C, the q-vector c, and the p-vector y using the technique of the previous section. Notice that these arrays are created under the assumption that the region of search is $[0, 1]^n$.
- LP-SOLVER(C, y, c, p, q) returns the q-vector u which minimizes the objective function $\mathbf{c}^T\mathbf{u}$ subject to the constraint $C\mathbf{u} = \mathbf{y}$.
- DOT-PRODUCT (c, u) returns the dot product of the two vectors c and u.
- SPAWN(a, b, f, n, sub) handles the recursive aspects of the algorithm. The array sub contains information about the convergence rate of the algorithm; if $sub[i]$ = TRUE, then the algorithm is converging too slowly in the ith coordinate direction. SPAWN first performs binary subdivision in each direction for which $sub[i]$ is TRUE (in other words, it splits the ith direction into two subintervals $[a[i], \frac{a[i]+b[i]}{2}]$ and $[\frac{a[i]+b[i]}{2}, b[i]]$). Thus, if there are exactly m TRUE values in the array sub, SPAWN will create 2^m subproblems by performing binary subdivision in each of the m directions. In the directions which do not require binary subdivision, simple subdivision according to a and b is performed. Hence if no binary subdivision is required on a given call to SPAWN, each f_k is subdivided so that f_k maps $[0, 1]^n$

to the portion of the function that at the beginning of execution was mapped to $[a[0], b[0]] \times [a[1], b[1]] \times \ldots \times [a[n], b[n]]$. Once these new boxes and functions are created, we call LP-RECUR once for each box and concatenate and return the resulting lists of points.

6.3 Explanation

We will now look at LP-FIND in more detail. The function LP-FIND itself is virtually the same as its counterpart PROJECTED-POLYHEDRON, so we will avoid repeating the explanation.

LP-RECUR becomes different from PP-RECUR at line 3. Here we call the function CREATE-BOX in order to allow us to call the linear-programming solver. The matrix and vectors which this routine creates are created exactly as described above. In lines 4–5 we allocate two arrays which will be used later: *cneg* and *sub*. Lines 6–7 initialize *cneg*; each element of *cneg* is the negative of the corresponding element of c. We need this auxiliary array because our linear programming solver LP-SOLVER (which, for example, could be an implementation of a simplex-type method) is configured to compute the minimum of the objective function. However, computing the minimum of *cneg* is equivalent to computing the maximum of c.

In line 9 we call the linear-programming solver in order to determine $a[i]$. If the special value NO-POINT is returned, then LP-SOLVER is telling us that the set A is empty, and therefore we return NO-POINT in line 11 as a signal to the calling routine that no root exists in our current search box. Otherwise, in line 12 we store the value of the objective function in the variable $a0$.

In line 13 we call the solver again, this time for the purpose of determining $b[i]$. The value of the objective function (maximum this time) is stored in $b0$.

In lines 15–18 we check to see how slowly the algorithm is converging. If the size of $b0 - a0$ is larger than CRIT (that is, fairly close to 1), then we set $sub[i]$ to TRUE in order to signify that binary subdivision is needed in that direction. Otherwise, $sub[i]$ is set to FALSE.

Lines 19–20 perform scaling on a0 and b0. Before we reached line 19, we had assumed that the box of search was $[0,1]^n$, when in fact it was the box determined by a and b. Now we scale $a0$ and $b0$ to resolve the discrepancy, and install the new values as the boundaries of the next box in lines 21–22. Finally, in line 23, we perform the recursive step of the algorithm by calling SPAWN, which we have already described.

6.4 Analysis

In this section, we will analyze the performance of the algorithm, just as we did with PROJECTED-POLYHEDRON. We will begin by defining a bounding-box tree similar to the one we introduced in section 5 and using it to prove that the algorithm terminates in a finite number of steps. Then we use some of the theorems in section 5 to show that LP-Find exhibits quadratic convergence.

6.4.1 The Bounding-Box Tree

As we shall see, the bounding-box tree for LP-FIND is similar to that of PROJECTED-POLYHEDRON. The major difference is that a given box may have as many as n children instead of a maximum of two. Our definitions of *step*, *bounding-box*, and *child* are slightly different from before, and so we will restate them:

Definition 6.3

1. *A step s is one execution of lines 1–22 of* LP-RECUR*. Note that there is only one step for each call to* LP-RECUR*.*
2. *A* bounding-box *B of a step s is the box defined by the n-dimensional arrays a and b at the beginning of the step.*

3. *A* child *of a bounding-box B of step s is one of the boxes which is passed onto a recursive call to* LP-RECUR *by* SPAWN *immediately following step s. Note that there may be as many as n children of B or none at all.*

The following lemma and corollary have completely analogous proofs to their counterparts in section 5, and so we state them without proof:

Lemma 6.4 *Let B_1 be a given node of the bounding-box tree, and suppose B_2 is one of its children. Then each side of B_2 is no larger than the corresponding side of B_1 times* CRIT.

Corollary 6.5 *Suppose we are given some number $\varepsilon > 0$ which is used in* SMALL-ENOUGH *as a tolerance. Then there exists an integer N such that after N or fewer steps, we generate a bounding-box tree such that all of the leaves are either boxes with each side smaller than ε or boxes which contain no root.*

Corollary 6.5 proves that the algorithm converges for any given ε tolerance after a finite number of steps. It turns out that this number is usually fairly small because LP-FIND exhibits quadratic convergence close to a root. The proof of this fact is the subject of section 6.4.3. But first, let us perform complexity analysis for the LP algorithm as we did for the PP algorithm.

6.4.2 Complexity Analysis

As before, we will assume that each polynomial is degree m in each variable, resulting in a total degree of mn for each of the n polynomials. Again, we do not know *a priori* how many steps will be required, but we can at least assess the cost of a call to LP-RECUR.

The cost of *LP-Solver* depends on the method used; so let us assume we are using the simplex method. This method operates on the $p \times q$ matrix C to generate a solution usually in $\mathcal{O}(p^2 q)$ time [13]. Assume that p is its largest possible value, $n^2 + n$; p will attain this value if all variables appear in all equations, and this is therefore a conservative estimate. As we shall see in section 7, there exist many examples in geometric modeling which involve sparse systems of equations and therefore result in a much smaller p. Since $q = (m+1)^n$, the total execution time of one call to *LP-Solver* in line 9 is $\mathcal{O}(n^4 m^n)$. LP-SOLVER is called again on the same size input in line 13, which does not affect the asymptotic running time. Since lines 8–22 are executed n times, the total cost of this loop is $\mathcal{O}(n^5 m^n)$.

Now, let us assume that SPAWN requires no binary subdivision in any direction. Then n subdivisions are required, resulting as before in a total time of $\mathcal{O}(n^2 m^{n+1})$. But what if some binary subdivision were required? We can get around this problem by *assessing the cost on the children of the current box.* Thus, if we generate a number of children on a given step, we charge the cost of generating each child to the child itself.

The total cost of one call to LP-RECUR is therefore $\mathcal{O}(n^5 m^n) + \mathcal{O}(n^2 m^{n+1})$ plus a possible term of $\mathcal{O}(n^2 m^{n+1})$ if the current box is being charged by its parent as a result of binary subdivision. Therefore, the total amortized cost for a step of LP-RECUR is $\mathcal{O}(n^5 m^n + n^2 m^{n+1})$. As we shall see in the next section, LP-FIND exhibits quadratic convergence; thus, usually LP-FIND needs fewer steps to finish than PROJECTED-POLYHEDRON. The smaller number of steps mitigates the extra n^2 factor we notice in comparing first terms of the LP and PP methods for small n. Furthermore, the LP method constructs tighter bounding boxes and is therefore likely to involve many fewer binary subdivision steps than the PP method.

6.4.3 Quadratic Convergence in n Dimensions

In this section, we will prove the following theorem:

Theorem 6.6 *Suppose we are given a box $B = [a_1, b_1] \times [a_2, b_2] \times [a_n, b_n]$ which contains one and only one root $\mathbf{x}_0$ of a system of polynomials $f_{0,k} : [0,1]^n \to \mathbf{R}$ where k ranges from 1 to n. Let f_k represent the* *subdivision of $f_{0,k}$ over B. Execute lines 1–22 of* LP-RECUR *once with these parameters, and let the new box at line 23 be denoted by $B' = [a'_1, b'_1] \times [a'_2, b'_2] \times [a'_n, b'_n]$. Define $\boldsymbol{\varepsilon} \equiv \mathbf{b} - \mathbf{a}$ and $\boldsymbol{\varepsilon}' \equiv \mathbf{b}' - \mathbf{a}'$. There exists a neighborhood U of $\mathbf{x}_0$ and a $\lambda > 0$ depending only on U such that if $B \subset U$ and* $\det \frac{\partial\{f_1, f_2, \ldots, f_n\}}{\partial\{x_1, x_2, \ldots, x_n\}} \neq 0$, *then* $\max_{k=1}^{n}\{\varepsilon'_k\} \leq \lambda \max_{k=1}^{n}\{\varepsilon_k^2\}$.

In simple terms, this theorem states that close to an isolated root, the algorithm exhibits quadratic convergence regardless of the dimension of the problem, obviously an improvement over the Projected-Polyhedron technique. As before, the proof involves linearization of the intersecting functions.

In the interest of saving space, we will use the informal $\mathcal{O}$ notation to describe the asymptotic properties of various quantities. Because this notation keeps us from thinking too much about detailed analysis, we can achieve greater clarity without sacrificing too much rigor.

Proof:

Step 1: In the interest of clarity, we will define ε to be $\max_{k=1}^{n}\{\varepsilon_k\}$. Let us start by linearizing all n of the f_k according to (28):

$$g_k(\mathbf{x}) = f_k(\mathbf{0}) + \sum_{j=1}^{n} x_j \frac{\partial f_k}{\partial x_j}(\mathbf{0}) \tag{66}$$

Suppose we determine the solution point in $\mathbf{R}^{n+1}$ to $g_1 = g_2 = \ldots = g_n = x_{n+1} = 0$. Now, if the determinant of the matrix $\frac{\partial\{f_1, f_2, \ldots, f_n\}}{\partial\{x_1, x_2, \ldots, x_n\}}$ is nonzero, then this system will have exactly one solution by Cramer's rule (the solution may not be in $[0,1]^n$, but we will not worry about that at the moment). Let us call this intersection point $\mathbf{p} = (p_1, p_2, \ldots, p_n, 0)$.

Step 2: Pick an arbitrary point $\mathbf{q} = (q_1, q_2, \ldots, q_n, 0)$ in the intersection set A defined previously. (Recall that A is the intersection of $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_n$ with the hyperplane $x_{n+1} = 0$.) Now, from (30) we know that

$$f_k(\mathbf{x}) - g_k(\mathbf{x}) = \mathcal{O}(\varepsilon^2) \tag{67}$$

for all $\mathbf{x} \in [0,1]^n$. Substituting $(q_1, q_2, \ldots, q_n)$ for $\mathbf{x}$, we see that since all f_k are zero at this point, g_k must be $\mathcal{O}(\varepsilon^2)$. Or equivalently, letting $r_k \equiv g_k(q_1, q_2, \ldots, q_n)$, there exist n points $\mathbf{r}_k = (q_1, q_2, \ldots, q_n, r_k)$ such that

1. $\mathbf{r}_k$ lies on the linear approximation to $\mathbf{F}_k$ and
2. r_k is an $\mathcal{O}(\varepsilon^2)$ quantity.

Step 3: Because $\mathbf{r}_k$ lies on the linear approximation to $\mathbf{F}_k$, it must be true that for each k,

$$r_k = f_k(\mathbf{0}) + \sum_{j=1}^{n} q_j \frac{\partial f_k}{\partial x_j}(\mathbf{0}). \tag{68}$$

Of course, since $\mathbf{p}$ is the intersection point of all of these linear approximations, it must also be true that for each k,

$$0 = f_k(\mathbf{0}) + \sum_{j=1}^{n} p_j \frac{\partial f_k}{\partial x_j}(\mathbf{0}). \tag{69}$$

Subtracting (69) from (68) for each k and defining $\delta_k \equiv q_k - p_k$, we obtain the system of k equations

$$\sum_{j=1}^{n} \delta_j \frac{\partial f_k}{\partial x_j}(\mathbf{0}) = r_k. \tag{70}$$

Assuming, again, that the Jacobian matrix $J = \frac{\partial\{f_1, f_2, \ldots, f_n\}}{\partial\{x_1, x_2, \ldots, x_n\}}$ is nonsingular, Cramer's rule enables us to find each δ_k matrix easily. According to the rule, if we let J_k be the matrix obtained by exchanging the kth column of J with the column vector $(r_1, r_2, \ldots, r_n)^T$, then

$$\delta_k = \frac{\det J_k}{\det J} \tag{71}$$

Recall (as in the proof of theorem 5.11) that the chain rule gives $\frac{\partial f_k}{\partial x_j} = \mathcal{O}(\varepsilon)$, and so each column of J has an element which is $\mathcal{O}(\varepsilon)$. But since every column of J is an $\mathcal{O}(\varepsilon)$ quantity, $\det J$ is $\mathcal{O}(\varepsilon^n)$. J_k has $n-1$ columns which are $\mathcal{O}(\varepsilon)$ and one column ($\mathbf{r}_k$) which is $\mathcal{O}(\varepsilon^2)$, and so $\det J_k$ is $\mathcal{O}(\varepsilon^{n+1})$. Thus, δ_k is $\mathcal{O}(\varepsilon)$.

Step 4: Since δ_k is $\mathcal{O}(\varepsilon)$ and $\mathbf{q}$ was chosen as an arbitrary element of A, we conclude that for each k, the kth coordinate of any point in A never deviates from a fixed value (i.e., p_k) by more than an $\mathcal{O}(\varepsilon)$ quantity. Therefore, by the definition of the LP bounding box, the length of each side of the box is an $\mathcal{O}(\varepsilon)$ quantity. But remember that the previous discussion has assumed that the input box is $[0,1]^n$ when in fact it is B. Scaling is therefore performed in lines 19–20; since the scaling factor is $b_k - a_k = \mathcal{O}(\varepsilon)$, we conclude that the resulting interval size

$$b'_k - a'_k = \mathcal{O}(\varepsilon)\mathcal{O}(\varepsilon) = \mathcal{O}(\varepsilon^2). \tag{72}$$

By applying the definition of the $\mathcal{O}$ notation, we conclude that there is some set U containing B (and thus B' and $\mathbf{x}_0$) such that the constant $\lambda > 0$ hidden within the $\mathcal{O}$ notation depends only on U. Translating (72) then gives $\varepsilon'_k = b'_k - a'_k \leq \lambda\varepsilon^2$, which in turn leads immediately to $\max_{k=1}^{n}\{\varepsilon'_k\} \leq \lambda \max_{k=1}^{n}\{\varepsilon_k^2\}$.□

This completes our analysis of the LP approach to root isolation. In the next section, we will give some practical problems, show how they can be reduced to systems of polynomial equations, and then solve them with both the PP and LP techniques.

7 Examples and Applications

In this section, we give a few numerical examples to demonstrate how well the techniques presented in this paper work. Then, we show how some other problems may be restated as systems of simultaneous polynomial equations and solved with one of the two algorithms. Finally, we include a running time analysis of our algorithms and provide some recommendations on choosing the most efficient algorithm.

7.1 Numerical Examples

The following six examples are designed to give the reader a better idea of the capabilities of the two algorithms. Implementation of both algorithms was in AT&T C++ version 2.0. C++ was chosen because both rational and floating point implementations could be written with little or no modification to the code.

7.1.1 Intersecting Two Simple Planar Curves

Our first example shows how to solve a simple planar intersection problem with both techniques. Suppose we want to intersect the circular arc

$$x^2 + y^2 - 1 = 0, 0 \leq x \leq 1, 0 \leq y \leq 1 \tag{73}$$

with the elliptical arc

$$\frac{x^2}{4} + 4y^2 - 1 = 0, 0 \leq x \leq 1, 0 \leq y \leq 1. \tag{74}$$

Simple algebra will show that these arcs intersect only in one point, $(\frac{2}{\sqrt{5}}, \frac{1}{\sqrt{5}})$, which to 9 digits is $(0.894427191, 0.447213595)$. In order to solve them with our methods, we first convert these two equations into the Bernstein basis, using the algorithm of appendix A. This results in the system

$$
\left[\begin{array}{ccc} b_{0,2}(x) & b_{1,2}(x) & b_{2,2}(x) \end{array}\right]\left[\begin{array}{ccc} -1 & -1 & 0 \\ -1 & -1 & 0 \\ 0 & 0 & 1 \end{array}\right]\left[\begin{array}{c} b_{0,2}(y) \\ b_{1,2}(y) \\ b_{2,2}(y) \end{array}\right] = 0 \tag{75}
$$

and

$$
\left[\begin{array}{ccc} b_{0,2}(x) & b_{1,2}(x) & b_{2,2}(x) \end{array}\right]\left[\begin{array}{ccc} -1 & -1 & 3 \\ -1 & -1 & 3 \\ \frac{3}{4} & \frac{3}{4} & \frac{13}{4} \end{array}\right]\left[\begin{array}{c} b_{0,2}(y) \\ b_{1,2}(y) \\ b_{2,2}(y) \end{array}\right] = 0. \tag{76}
$$

We can now directly apply either method to this system. Running the Projected-Polyhedron algorithm gives us the following results

Bounding Boxes from Projected-Polyhedron					
Iter	a_1	b_1	a_2	b_2	*Message*
1	0	1	0	1	Binary subdivision in x
2	0	0.5	0	1	No root in box
3	0.5	1.0	0.1875	0.61718750	None
4	0.74605306	0.98242188	0.37952697	0.48294336	None
5	0.86764611	0.92684826	0.44013056	0.45158886	None
6	0.89175192	0.89838481	0.44668154	0.44756546	None
7	0.89424540	0.89469787	0.44717954	0.44723644	None
8	0.89441574	0.89444424	0.44721146	0.44721503	None
9	0.89442648	0.89442826	0.44721346	0.44721368	None
10	0.89442715	0.89442726	0.44721359	0.44721360	None
11	0.89442719	0.89442719	0.44721360	0.44721360	Root Found

This particular example was run at a tolerance of 10^{-8}. Notice the linear convergence rate as we approach the root.

Now let us turn to the LP algorithm. On the first iteration, the LP routine will set up the constraint matrix C as follows:

$$
\left[\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 1 & 1 & 1 & 0 & 0 & 0 & -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & -1 & -1 & -1 \\
0 & \frac{1}{2} & 1 & 0 & \frac{1}{2} & 1 & 0 & \frac{1}{2} & 1 & 0 & -\frac{1}{2} & -1 & 0 & -\frac{1}{2} & -1 & 0 & -\frac{1}{2} & -1 \\
-1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & -3 & 1 & 1 & -3 & \frac{3}{4} & \frac{3}{4} & -\frac{13}{4} \\
-1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}\right] \tag{77}
$$

and the vector of right-hand-sides $\mathbf{r}$ will be

$$
\mathbf{r}^T = \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right] \tag{78}
$$

The vector of unknown convex hull coefficients $\mathbf{u}$ is constrained to satisfy $C\mathbf{u} = \mathbf{r}$. Now if, for example, we want to find a_1, the smallest x-coordinate of the bounding box, then we simply find the minimum value of $\mathbf{c}^T\mathbf{u}$ where $\mathbf{c}$ equals

$$
\left[\begin{array}{cccccccccccccccccc} 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]. \tag{79}
$$

b_1, a_2, and b_2 may be found similarly, with the same matrix C and right-hand side $\mathbf{r}$, but with different coefficient vectors $\mathbf{c}$. If we let the algorithm continue, we observe the following results:

Bounding Boxes $[a_1, b_1] \times [a_2, b_2]$ from LP-Find					
Iter	a_1	b_1	a_2	b_2	*Message*
1	0	1	0	1	None
2	0.375	1	0.1875	0.625	None
3	0.80012077	0.94619865	0.38069053	0.48248792	None
4	0.88925091	0.89831844	0.44364954	0.44931108	None
5	0.89440075	0.89444401	0.44720171	0.44722200	None
6	0.89442719	0.89442719	0.44721360	0.44721360	Root Found

These results were obtained with a tolerance of 10^{-8}. Notice how this example illustrates the quadratic convergence of the linear-programming approach; the number of significant digits of each coordinate approximately doubles at each step.

7.1.2 The Wilkinson Polynomial

Our next example attempts to solve the following twentieth degree polynomial which was shown by Wilkinson [32] to be extremely ill-conditioned:

$$P_{20}(x) = (x-1)(x-2)\dots(x-20). \tag{80}$$

In [10], it is shown that the roots of this polynomial may be extracted with much less error by considering the polynomial in the Bernstein basis before attempting solution. In accordance with this approach, we first scale the polynomial so that all 20 of the roots lie within $[0, 1]$, i.e.

$$P(x) = \prod_{k=1}^{20}(x - \frac{k}{20}). \tag{81}$$

We begin by converting this polynomial into the Bernstein basis, as shown in Appendix A. Some care is needed here, because although once the polynomial has been transformed it is fairly stable, nevertheless the conversion process itself is ill-conditioned. To eliminate this problem, we start with exact rational coefficients and perform the conversion in exact rational arithmetic; once the process has been completed, we can easily convert the rational Bernstein coefficients into floating-point numbers with accuracy on the order of machine precision [16]. Since this is only a one-dimensional problem, both the PP and the LP techniques are quadratically convergent. Given a tolerance of 10^{-7}, the LP technique generated the following results:

Roots of $P(x)$		
Root number	root	error
1	0.0500000002437804	2.4×10^{-10}
2	0.10000000000016	1.6×10^{-13}
3	0.150000000000427	4.3×10^{-13}
4	0.200000000000014	1.4×10^{-14}
5	0.249999999999785	2.2×10^{-13}
6	0.300000000009474	9.5×10^{-12}
7	0.350000000092444	9.2×10^{-11}
8	0.400000000006641	6.6×10^{-12}
9	0.450000024696511	2.5×10^{-8}
10	0.499999980911975	1.9×10^{-8}
11	0.549999999963676	3.6×10^{-11}
12	0.599999984022245	1.6×10^{-8}
13	0.649999993324551	6.7×10^{-9}
14	0.699999993937085	6.1×10^{-9}
15	0.749999999997894	2.1×10^{-12}
16	0.79999999503061	5.0×10^{-9}
17	0.849999968721786	3.1×10^{-8}
18	0.899999999995995	4.0×10^{-12}
19	0.949999989125186	1.1×10^{-8}
20	1.00000000000001	1.0×10^{-14}

As the table indicates, both the linear programming and subdivision phases were stable enough to find all roots of this extremely unstable polynomial.

7.1.3 Significant Points of a Planar Algebraic Curve

A well known problem in computer-aided design is the discovery and subsequent tracing of all branches of an implicit algebraic curve $f(u,v) = 0$. The computation of *turning points* (where $f = \frac{\partial f}{\partial u} = 0$ or $f = \frac{\partial f}{\partial v} = 0$) or *critical points* (where $\frac{\partial f}{\partial u} = \frac{\partial f}{\partial v} = 0$) is important in solving this problem [23] [27] [17] [26]. As an example, let us find the critical points of the curve $f(u,v) = 0$ illustrated in figure 4, which is defined by

$$\begin{aligned} f(u,v) = -64v^4 + 128v^3 - 96u^2v^2 + 140uv - 139v^2 + 96u^2v - 140uv + 75v \\ -96u^4 + 276u^3 - 313u^2 + 165u - 36 = 0 \end{aligned} \tag{82}$$

Taking the two partial derivatives gives us two simultaneous equations which we can use either method to solve. With a tolerance of 10^{-8}, the LP method computes the following nine roots:

Critical points of f to 8 digits		
Root number	u	v
1	0.92143996	0.50000000
2	0.83640890	0.73280861
3	0.75000000	0.75000000
4	0.73481004	0.50000000
5	0.55109110	0.69759134
6	0.50000000	0.50000000
7	0.83640890	0.26719139
8	0.75000000	0.25000000
9	0.55109110	0.30240866

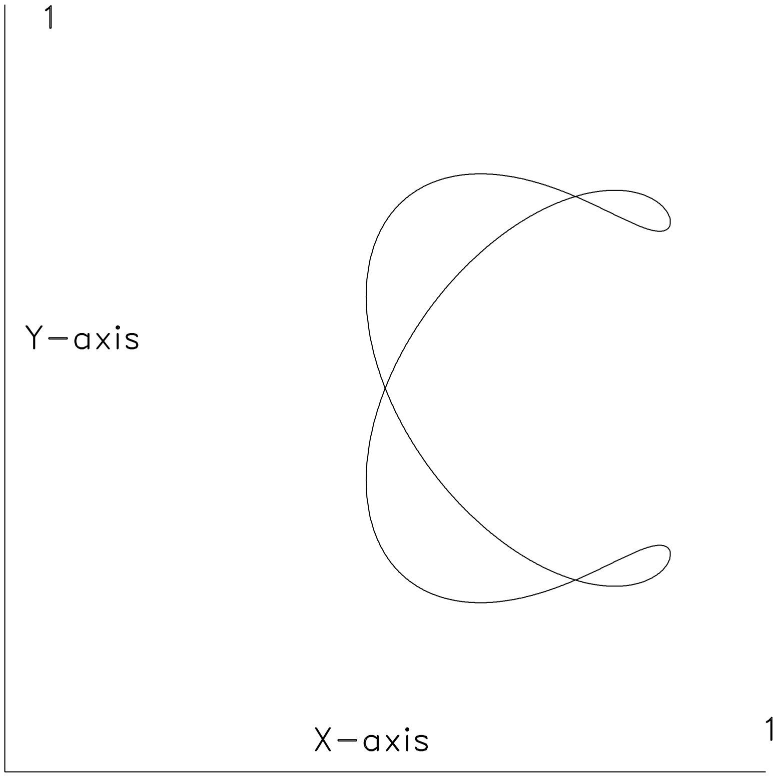

Figure 4: The curve $f(u, v) = 0$ of section 7.1.3

7.1.4 Stationary Points of a Distance Function

Our next example is essentially a problem of finding extrema. Suppose we are given two implicit algebraic surfaces $f(x, y, z) = 0$ and $g(x, y, z) = 0$. Now consider all possible pairs of points $\mathbf{p}_1 = (x_1, y_1, z_1)$ and $\mathbf{p}_2 = (x_2, y_2, z_2)$, where $\mathbf{p}_1$ lies on f and $\mathbf{p}_2$ lies on g. Suppose we want to find those pairs of points which are farthest apart, or closest together. Then clearly these points are extrema of the square of the distance metric

$$D = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 \tag{83}$$

As an example, let

$$f(x_1, y_1, z_1) = (x_1 - \frac{1}{5})^2 + (y_1 - \frac{1}{5})^2 + (z_1 - \frac{1}{5})^2 - \frac{1}{25} = 0 \tag{84}$$

$$g(x_2, y_2, z_2) = (x_2 - \frac{1}{5})^2 + (y_2 - \frac{1}{5})^2 + (z_2 - \frac{4}{5})^2 - \frac{1}{25} = 0. \tag{85}$$

We will consider x_1, y_1, x_2, and y_2 to be independent variables, while z_1 depends on x_1 and y_1 and z_2 depends on x_2 and y_2. Now, for $(x_1, y_1, z_1, x_2, y_2, z_2)$ to be an extremum, the partial derivatives of D with respect to the independent parameters must be 0. Accordingly, consider x_1, y_1, x_2, and y_2 to be independent, and take partial derivatives of D with respect to these four variables. We obtain

$$\frac{\partial D}{\partial x_1} = 2(x_1 - x_2) + 2(z_1 - z_2)\frac{\partial z_1}{\partial x_1} = 0 \tag{86}$$

and similar expressions for the other three derivatives. Implicit differentiation of (84) and (85), gives us

$$\frac{\partial z_1}{\partial x_1} = -\frac{x_1 - \frac{1}{5}}{z_1 - \frac{1}{5}} \tag{87}$$

and similar expressions for the partial derivatives $\frac{\partial z_1}{\partial y_1}$, $\frac{\partial z_2}{\partial x_2}$, and $\frac{\partial z_1}{\partial y_1}$. Substituting these four equations into the partial derivatives for D and simplifying gives

$$(x_1 - x_2)(z_1 - \frac{1}{5}) - (x_1 - \frac{1}{5})(z_1 - z_2) = 0 \tag{88}$$

$$(y_1 - y_2)(z_1 - \frac{1}{5}) - (y_1 - \frac{1}{5})(z_1 - z_2) = 0 \tag{89}$$

$$(x_2 - x_1)(z_2 - \frac{4}{5}) - (x_2 - \frac{1}{5})(z_2 - z_1) = 0 \tag{90}$$

$$(y_2 - x_1)(z_2 - \frac{4}{5}) - (y_2 - \frac{1}{5})(z_2 - z_1) = 0 \tag{91}$$

These four equations together with (84) and (85) form a system of six polynomial equations in six unknowns, which may now be converted to Bernstein form and solved with the PP technique.

The PP technique yields the following four roots when run at a tolerance of 10^{-8}:

Critical points of the distance function to 8 digits						
Root number	x_1	y_1	z_1	x_2	y_2	z_2
1	0.2	0.2	0	0.2	0.2	1
1	0.2	0.2	0.4	0.2	0.2	1
1	0.2	0.2	0	0.2	0.2	0.6
1	0.2	0.2	0.4	0.2	0.2	0.6

7.1.5 A Stationary Point Problem at a Strict Tolerance

Consider the function

$$f(x,y) = [(x-\frac{1}{4})^3 + (y-\frac{1}{4})^3 - (\frac{3}{5})^3]^2 + [(x-\frac{3}{4})^3 + (y-\frac{3}{4})^3 - (\frac{3}{5})^3]^2 \tag{92}$$

and set its partial derivatives equal to zero. Solving the resulting system at a strict tolerance of 10^{-14} gives one real root at $(0.72660262158698, 0.72660262158698)$.

7.1.6 A Distance Problem with Tangency

Our last example illustrates a problem with our techniques due to tangency. Consider the first quadrant of the unit circle $x^2+y^2 = 1$ and the portion of the parabola $y = 1-x^2$ lying above the x-axis, both expressed as parametric rational Bézier curves. We wish to compute the stationary points of the squared distance between the two curves; accordingly, we set $D(t,u) = (x_1 - x_2)^2 + (y_1 - y_2)^2$ (where $(x_1(t), y_1(t))$ is the parametric equation for the parabola and $(x_2(u), y_2(u))$ is the parametric equation for the circle), take partial derivatives with respect to t and u, and set them equal to zero. Because these curves are both assumed to be rational, we end up with one equation of degree 8 in t and degree 9 in u, and one equation of degree 8 in u and degree 9 in t. In this example, because there is a tangency at $(0,1)$ in the (x,y)-plane and the curves are nearly parallel, the rate of convergence drops and an extensive amount of binary subdivision is performed; consequently, this example takes a fairly long time to run to a tolerance of 10^{-8}. This example has the following pairs of minimum distance points: $(0,1)$ on the circle and $(0,1)$ on the parabola; $(1,0)$ and $(1,0)$; $(\frac{\sqrt{2}}{\sqrt{3}}, \frac{1}{\sqrt{3}})$ and $(\frac{1}{\sqrt{2}}, \frac{1}{2})$.

7.2 Other Applications

In closing this section, we will briefly touch on some other applications of our algorithms and how they might be used or extended to solve related problems.

7.2.1 Systems of Piecewise Polynomials

Up to this point, we have said nothing about extending this algorithm to piecewise polynomials. However, since B-splines possess the convex hull property, such an extension should be fairly straightforward. One simple way to solve such a problem, of course, would be to split the piecewise polynomials into separate Bernstein polynomials [2] [5] in order to solve a number of different polynomial subproblems. But since both the PP and LP algorithms rely only the convex hull property to generate new bounding boxes, reformulating the problem in the Bernstein basis is unnecessary.

Problems analogous to the four we have examined above are often encountered with piecewise polynomial entities. Determining the stationary points of the distance function between two piecewise polynomial entities may seem difficult because multiplication of piecewise polynomials is required; however, recent research by Morken [19] has shown that this is a tractable problem. Whether or not this multiplication of piecewise polynomials represents a significant improvement over subdivision into Bernstein components and then multiplication of Bernstein polynomials remains to be studied.

7.2.2 More Equations than Unknowns, or Vice-Versa

Throughout this paper, we have been assuming that the number of equations is equal to the number of unknowns, mostly because the problem is easier to think of in this form. However, there is no particular reason that the two numbers must be equal. One common example is determining the singular points of an implicit algebraic curve $f(u,v) = 0$. These points are the solutions of the system $f = \frac{\partial f}{\partial u} = \frac{\partial f}{\partial v} = 0$, which involves three equations in two unknowns. Solving this problem requires nothing more than a few more constraints in the LP approach, or one more polyhedron to project in the PP method.

Fewer equations than unknowns should also be possible to solve. For example, such problems arise in engineering design of complex structures.

7.2.3 Finding Complex Roots

These techniques can easily be applied to find complex roots as well as real roots. Simply substitute $\mathbf{u} + i\mathbf{v}$ for $\mathbf{x}$ in (8), where i is the imaginary unit, and separate the real and imaginary parts. In this formulation, the coefficients $w_I^{(k)}$ may be complex. In general, for n simultaneous complex polynomial equations, one obtains $2n$ real equations.

7.3 Choosing an Algorithm

Looking at the complexity estimates per iteration of sections 5.4.2 and 6.4.2 gives a clear indication of which algorithms perform better under various circumstances. There are two cases to consider. If $n^3 \leq m$, the second term of the LP complexity estimate is more important, and thus both the LP and PP algorithms have the same asymptotic time of $\mathcal{O}(n^2 m^{n+1})$. Otherwise, the PP is faster by a factor of $\frac{n^3}{m}$. Here n is the number of variables, m is the degree of each variable in each equation. The LP algorithm will therefore be better when n is small and when the number of iterations required is considerably less. Because the LP algorithm exhibits quadratic convergence near simple roots, we expect it to perform exceptionally well when the tolerance is fairly strict. This is in fact exactly what we observe in practice. In addition, numerical experiments suggest that for small n, the LP is more efficient than the PP for multiple roots.

However, if the problem in question has a large number of roots in the domain, both algorithms are forced to perform a large number of subdivision steps in the early stages of the algorithm; in such cases, when many iterations are needed, the PP algorithm tends to perform better. Since we may not know *a priori* how many roots a given problem will have, a *hybrid algorithm* based on both methods is useful for low n. In this approach, we perform a number of fast PP iterations until the sizes of the bounding boxes are smaller than a certain coarse tolerance (for example, 10^{-3}). After the boxes drop below this size, we employ a few LP iterations in order to take advantage of quadratic convergence.

This hybrid approach typically performs well for most $n = 2$ and some $n = 3$ problems. However, for larger n, LP iterations become too expensive, and we recommend a pure PP approach for such cases.

These observations are reflected in the following table. The timings correspond to results obtained on a graphics workstation running IRIX 4.0.4 at 36 MHz. In this table, the PP, LP and hybrid (HY) algorithms are all tested. For the hybrid algorithm, the PP was run at a tolerance of 10^{-4} before beginning LP iterations. Under the time tests is a row indicating the tolerance at which the examples were run; the examples themselves are numbered according to their section. For instance example number 2 is the Wilkinson polynomial of section 7.1.2. An entry of * in the table means that the example executed too quickly to be measured.

CPU timings (in seconds)						
	1	2	3	4	5	6
PP	*	0.1	0.4	4.2	1.7	23000
LP	*	0.2	1.1	225	0.7	1810
HY	*	0.2	0.4	65	0.5	1940
Tol	10^{-8}	10^{-7}	10^{-8}	10^{-8}	10^{-14}	10^{-8}

8 Concluding Remarks

Both the Linear Programming and Projected-Polyhedron algorithms were implemented with exact rational arithmetic as well as double-precision floating point. The rational arithmetic library was written in C++, based on the algorithms for arbitrary-precision arithmetic given in Knuth [16].

We decided to write versions in rational arithmetic as well as floating point for several reasons. First, we noticed that floating point implementations tended to perturb roots unpleasantly or miss them altogether in ill-conditioned problems when the tolerance went below 10^{-8} or so. With arbitrary-precision rational arithmetic, of course, this was not a problem; the lower bound on the accuracy of roots is determined only by the memory of the machine, and so any reasonable accuracy may theoretically be achieved. Secondly, we wanted to see under what conditions rational arithmetic could achieve results in relatively short periods of time.

What we found was that although it was extremely useful (as well as fast) in the preprocessing necessary to convert equations from the power to the Bernstein basis, in the solution phase rational arithmetic was rarely fast enough for practical application. The results are most dramatic in the Wilkinson polynomial example of section 7. Although it runs almost instantaneously in floating point, it takes on the order of minutes to execute using rationals. This is because performing De Casteljau subdivision on a twentieth degree polynomial results in many additions, which make the numerators and denominators of the rationals huge and time-consuming to process.

On the other hand, the rational implementation performed quite well with moderate-degree polynomials, since the subdivision phase resulted in much shorter (and therefore more tractable) coefficients. In order to keep them shorter, however, it was necessary to modify the bounding boxes slightly after they were generated. Due to a number of rational operations, these boxes would often have components with moderate-sized numerators and denominators. To rectify this problem, we would create a slightly larger bounding box with much shorter components and use this box on the next iteration.

There is some question as to whether rationals are really needed in our procedures. Both of the algorithms, especially the Linear-Programming technique, are inherently well-conditioned, and did not suffer unduly from floating-point error. If higher accuracy is desired, a possibility would be to study the use of extended precision floating point arithmetic rather than arbitrary-precision (i.e. more bits in the mantissa than standard double-precision). This approach could preserve much of the speed of floating point while achieving higher accuracy. The development of methods which allow adaptive selection of appropriate precision for a given problem is recommended for futher study.

A few modifications to the algorithms could improve performance. For example, either method might be altered to intersect polynomials expressed in terms of barycentric coordinates [9]. This has already been done for $n = 2$ in [28] using projections of convex hulls. Another improvement would be the ability to select, by examing the pattern of bounding-box reduction, a better place than the midpoint to split a box when convergence slows. Finally, more examples need to be run on fully optimized versions of the two algorithms. The resulting comparisons between the two and between other methods would help in choosing an algorithm to fit a given class of applications.

Acknowledgments

This work was supported, in part, by the M.I.T. Sea Grant College Program and the Office of Naval Research under grant numbers NA90AA-D-SG-424 and N00014-91-J-1014. We wish to thank Dr. Craig C. Sherbrooke for valued assistance on the theory of Linear Programming. We would also like to thank the reviewers for their comments and Professor T. W. Sederberg for identifying relevant references.

Authors' Addresses

Mr. Evan C. Sherbrooke
M.I.T. Room 5-424, Cambridge, MA 02139-9910, USA
Tel: (617)253-7950 Fax: (617)253-8125 E-MAIL: esher@athena.mit.edu

Prof. Nicholas M. Patrikalakis
M.I.T. Room 5-428, Cambridge, MA 02139-9910, USA
Tel: (617)253-4555 Fax: (617)253-8125 E-MAIL: nmp@deslab.mit.edu

A Converting from the Power to the Bernstein Basis

In this appendix we present some pseudocode which demonstrates how to convert a polynomial in n variables from the power basis to the Bernstein basis. On input, we assume that we are given an array m indexed from 1 to n and an n-dimensional array a of power basis coefficients indexed in the ith dimension from 0 to $m[i]$. The Bernstein coefficients are stored in the identically sized n-dimensional array b.

This particular algorithm is a simple adaptation to polynomials in n variables of the algorithm for converting univariate polynomials given in [10]. The extension is straightforward, so we omit the derivation for the sake of brevity.

In the code below, we have glossed over a few important details of implementation for the sake of clarity; for example, no mainstream computer language will actually allow the expression $a[j_1][j_2]\ldots[j_n]$ to index an n-dimensional array unless n is fixed at compile-time. (However, this problem may be resolved by using one-dimensional arrays instead of n-dimensional arrays. For example, a three-dimensional array a with elements ranging from $a[0][0][0]$ to $a[m_1][m_2][m_3]$ could be stored as a one dimensional array b of $(m_1+1)(m_2+1)(m_3+1)$ elements. With this scheme, $a[i][j][k]$ would be accessed by referencing $b[i(m_2+1)(m_3+1)+j(m_3+1)+k]$. This is the sort of approach most computer languages take in addressing multidimensional arrays.)

Power-to-Bern (a, b, m, n)
for $i \leftarrow 1$ to n
    for $(j_1, j_2, \ldots, j_{i-1}, j_{i+1}, \ldots, j_n) \leftarrow (0, 0, \ldots, 0)$ to $(m[1], m[2], \ldots, m[i-1], m[i+1], \ldots, m[n])$
        for $k \leftarrow 1$ to $m[i]$
            $sum \leftarrow 0$

5 for $l \leftarrow 1$ to $m[i]$
6 $\quad$ $sum \leftarrow sum + a[j_1][j_2]\ldots[j_{i-1}][l][j_{i+1}]\ldots[j_n]\textsc{Binomial}(k,l)/\textsc{Binomial}(m[i],l)$
7 $b[j_1][j_2]\ldots[j_{i-1}][k][j_{i+1}]\ldots[j_n] \leftarrow sum$

The for loop in line 2 indicates that we loop over all possible multi-indices $(j_1, j_2, \ldots, j_{i-1}, j_{i+1}, \ldots, j_n)$ which are bounded by the multi-index $(m[1], m[2], \ldots, m[i-1], m[i+1], \ldots, m[n])$.

B Subdivision of a Multivariate Bernstein Polynomial

In this appendix we give pseudocode which will allow us to subdivide a polynomial f_0 of n variables $x_1, x_2, \ldots, x_n$. We assume that on input we are given two arrays of real numbers a and b which define a box $B = [a[1], b[1]] \times [a[2], b[2]] \times \ldots \times [a[n], b[n]] \subset [0,1]^n$. The act of subdivision will allow us to create a new polynomial f which is defined by

$$f(x_1, x_2, \ldots, x_n) = f_0\{a[1] + (b[1] - a[1])x_1, a[2] + (b[2] - a[2])x_2, \ldots, a[n] + (b[n] - a[n])x_n\} \tag{93}$$

We assume that f_0 and f are n-dimensional arrays indexed in the ith dimension from 0 to $m[i]$, where m is an array indexed from 1 to n. The method of subdivision is De Casteljau subdivision, and the algorithm below is an extension of similar algorithms for 1 and 2 dimensions given in [9]. Because of this similarity, we do not bother to give a proof that it works.

DC-Box-Subdivide (f_0, a, b, m, n, f)
for $i \leftarrow 1$ to n
if $b[i] = 0$ $a0 \leftarrow 0$ else $a0 \leftarrow a[i]/b[i]$
$b0 \leftarrow b[i]$
$\quad$ $r \leftarrow$ 2D-Array $(0, m[i], 0, m[i])$
$\quad$ for $(j_1, j_2, \ldots, j_{i-1}, j_{i+1}, j_n) \leftarrow (0, 0, \ldots, 0)$ to $(m[1], m[2], \ldots, m[i-1], m[i+1], \ldots, m[n])$
$\quad\quad$ for $k \leftarrow 1$ to $m[i]$
$\quad\quad\quad$ $r[k][0] \leftarrow f_0[j_1][j_2]\ldots[j_{i-1}][k][j_{i+1}]\ldots[j_n]$
$\quad\quad$ for $k \leftarrow 1$ to $m[i]$
$\quad\quad\quad$ for $l \leftarrow k$ to $m[i]$
$\quad\quad\quad\quad$ $r[l][k] \leftarrow (1-b)r[i-1][k-1] + b\ r[i][k-1]$
$\quad\quad$ for $k \leftarrow 1$ to $m[i]$
$\quad\quad\quad$ $r[k][0] \leftarrow r[k][k]$
$\quad\quad$ for $k \leftarrow 1$ to $m[i]$
$\quad\quad\quad$ for $l \leftarrow k$ to $m[i]$
$\quad\quad\quad\quad$ $r[l][k] \leftarrow (1-a)r[i-1][k-1] + a\ r[i][k-1]$
$\quad\quad$ for $k \leftarrow 1$ to $m[i]$
$\quad\quad\quad$ $f[j_1][j_2]\ldots[j_{i-1}][k][j_{i+1}]\ldots[j_n] \leftarrow r[m[i]][m[i]-k]$

References

[1] C. Bliek. *Computer Methods for Design Automation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1992.

[2] W. Boehm. Inserting new knots into B-spline curves. *Computer Aided Design*, 12(4):199–201, 1980.

[3] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory: Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985. Dordrecht, Holland: D. Reidel Publishing Company.

[4] J. Canny. Generalized characteristic polynomials. *Journal of Symbolic Computation*, 9:241–250, 1990.

[5] E. Cohen, T. Lyche, and R. 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.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[7] G. Dahlquist and A. Björck. *Numerical Methods.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.

[8] T. Dokken. Finding intersections of B-spline represented geometries using recursive subdivision techniques. *Computer Aided Geometric Design*, 2:189–195, 1985.

[9] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design- A Practical Guide.* Academic Press, New York, 2nd edition, 1990.

[10] R. T. Farouki and V. T. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4:191–216, March 1987.

[11] R. T. Farouki and V. T. Rajan. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design*, 5:1–26, 1988.

[12] C. B. Garcia and W. I. Zangwill. Global continuation methods for finding all solutions to polynomial systems of equations in n variables. In A. V. Fiacco and K. O. Kortanek, editors, *Extremal Methods and Systems Analysis*, pages 481–497. Springer-Verlag, New York, NY, 1980.

[13] S. I. Gass. *Linear Programming: Methods and Applications.* McGraw-Hill, New York, 1985.

[14] A. Geisow. *Surface Interrogations.* PhD thesis, School of Computing Studies and Accountancy, University of East Anglia, Norwich NR47TJ, U. K., July 1983.

[15] R. B. Kearfott. Decomposition of arithmetic expressions to improve the behavior of interval iteration for nonlinear systems. *Computing*, 47:169–191, 1991.

[16] D. E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 1981. 2nd Edition.

[17] G. A. Kriezis, P. V. Prakash, and N. M. Patrikalakis. A method for intersecting algebraic surfaces with rational polynomial patches. *Computer Aided Design*, 22(10):645–654, December 1990.

[18] J. M. Lane and R. F. Riesenfeld. Bounds on a polynomial. *BIT: Nordisk Tidskrift for Informations-Behandling*, 21(1):112–117, 1981.

[19] K. Mørken. Some identities for products and degree raising of splines. *Constructive Approximation*, 7:195–208, 1991.

[20] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, 1990.

[21] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *ACM Computer Graphics*, 24(4):337–345, August 1990.

[22] N. M. Patrikalakis and G. A. Kriezis. Representation of piecewise continuous algebraic surfaces in terms of B-splines. *The Visual Computer*, 5(6):360–374, 1989.

[23] N. M. Patrikalakis and P. V. Prakash. Surface intersections for geometric modeling. *Journal of Mechanical Design, ASME Transactions*, 112(1):100–107, March 1990.

[24] P. V. Prakash and N. M. Patrikalakis. Surface-to-surface intersections for geometric modeling. Technical Report MITSG 88-8, Cambridge, MA: MIT Sea Grant College Program, 1988.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[26] T. Sakkalis. The topological configuration of a real algebraic curve. *Bulletin of the Australian Mathematical Society*, 43:37–50, 1991.

[27] T. Sakkalis and R. T. Farouki. Singular points of algebraic curves. *Journal of Symbolic Computation*, 9:405–421, 1990.

[28] T. W. Sederberg. Algorithm for algebraic curve intersection. *Computer Aided Design*, 21(9):547–554, November 1989.

[29] T. W. Sederberg and T. Nishita. Geometric Hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

[30] Symbolics, Inc. *MACSYMA Reference Manual*, version 13 edition, November 1988.

[31] M. E. Vafiadou and N. M. Patrikalakis. Interrogation of offsets of polynomial surface patches. In F. H. Post and W. Barth, editors, *Eurographics '91, Proceedings of the 12th Annual European Association for Computer Graphics Conference and Exhibition*, pages 247–259 and 538, Vienna, Austria, September 1991. Amsterdam: North-Holland.

[32] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* H. M. Stationery Office, London, England, 1963.

[33] W. I. Zangwill and C. B. Garcia. *Pathways to solutions, fixed points, and equilibria.* Prentice-Hall, Englewood Cliffs, NJ, 1981.