PRECISE: Efficient Multiprecision Evaluation of Algebraic Roots and Predicates for Reliable Geometric Computation

Shankar Krishnan
AT&T Labs - Research
krishnas@research.att.com

Mark Foskey
Univ. of North Carolina
foskey@cs.unc.edu

Tim Culver
Think3
culver@acm.org

John Keyser
Texas A&M Univ.
keyser@cs.tamu.edu

Dinesh Manocha
Univ. of North Carolina
dm@cs.unc.edu

Abstract:
Many geometric problems like generalized Voronoi diagrams, medial axis computations and boundary evaluation involve computation and manipulation of non-linear algebraic primitives like curves and surfaces. The algorithms designed for these problems make decisions based on signs of geometric predicates or on the roots of polynomials characterizing the problem. The reliability of the algorithm depends on the accurate evaluation of these signs and roots. In this paper, we present a *naive precision-driven computational model* to perform these computations reliably and demonstrate its effectiveness on a certain class of problems like sign of determinants with rational entries, boundary evaluation and curve arrangements. We also present a novel algorithm to compute all the roots of a univariate polynomial to any desired accuracy. The computational model along with the underlying number representation, precision-driven arithmetic and all the algorithms are implemented as part of a stand-alone software library, PRECISE.

1. INTRODUCTION

Many geometric algorithms make decisions based on signs of geometric predicates. In most cases, these predicates correspond to algebraic functions of input parameters. The reliability of the algorithm depends on the accurate evaluation of the signs of the predicates. A common example is classifying the location of a point with regard to some other geometric object, which could be a line, plane, circle, sphere, or even a closed region bounded by piecewise algebraic curves or surfaces.

Conceptually, the simplest solution to the problem of determining accurate geometric predicates involves the use of exact arithmetic. Such arithmetic is based on arbitrary precision representation of intermediate and final expressions, with arithmetic implemented in software. Direct use of these algorithms can be quite slow and hence many techniques have been proposed to improve their speed. One approach involves explicitly keeping track of error in the expression evaluation [26, 42, 3]. Alternatively, others have developed improved algorithms for arbitrary precision arithmetic [38, 22, 7, 36, 4]. Finally, it is possible to use filters that exploit features of the predicates or make assumptions on the maximum precision needed to evaluate a predicate [20, 38, 43]. These algorithms have been successfully used for computing convex hulls and Voronoi diagrams of point sets and Boolean combinations of polyhedral primitives.

Many geometric applications involve computation and manipulation of non-linear algebraic primitives. These include generalized Voronoi diagrams and medial axis computation, bisector surfaces, boundary evaluation of algebraic primitives and Minkowski sums of polyhedra. In such cases, the basic primitives like points, curves and surfaces are represented using algebraic numbers and polynomial equations. Techniques for exact representation of these primitives have been proposed [11, 28, 15]. The number of bits required for an exact representation can grow significantly with the algebraic degree[1]. Given an accurate representation of the primitives, the underlying computations that have to be performed in these applications are:

- Isolating and evaluating the roots of a polynomial system.
- Accurately evaluating the sign of an algebraic expression, such as a matrix determinant.

One of the commonly used approaches for root isolation is based on multi-variate Sturm sequences [35, 27, 28, 15]. However, these techniques tend to be slow because of coefficient growth when computations involving moderate degree algebraic primitives are performed. Another approach is to approximate the roots of the system using numerical iterative techniques. These approaches can suffer from convergence problems due to ill-conditioning or clustering of roots, and they may not provide sufficient accuracy in any event.

Recently, research has focused on *precision-driven computation* [32, 9, 10, 25, 31], where exact computation is performed on expressions involving rational operations, k^{th} root operation and comparisons by maintaining directed acyclic graphs of expressions, or *expression dags* (*leda_real* [10] and *Expr* [25]). However, when the desired computation cannot be conveniently expressed in closed form, it is not clear how this approach can be applied. For instance,

[1]The worst case bounds are exponential in the algebraic degree

SCG'01, June 3-5, 2001, Medford, Massachusetts, USA.

the closed form expression of a matrix determinant quickly becomes prohibitively complex as the degree of the matrix increases. As another example, polynomial root finding is often best performed by iterative algorithms for which it is impossible to construct an expression dag in advance. We are interested in problems that fall into this category. We believe that the right approach for these computations is to determine precision by *iterative revision*. In essence, our approach is one of trial and error: We perform a calculation at a specified precision, keeping track of the loss of accuracy by a kind of interval arithmetic. If the accuracy of the result is insufficient, we increase the precision of our representation and redo the calculation.

1.1 Main Results

In this paper, we present representations and algorithms that can reliably perform computations not given in closed form, returning results to an accuracy chosen in advance. We demonstrate its effectiveness on a variety of problems including determinant sign evaluation, curve arrangements, boundary evaluation, curve arrangements, and medial axis computation. We also present a novel algorithm to compute all the roots of a univariate polynomial to any desired precision.

Our number type (*real*) is based on extended precision floating point, but it also maintains an explicit error interval of fixed length. Efficient algorithms for all the basic arithmetic operations, k^{th} root operator and transcendental functions are performed on top of the above number representation. The underlying number representation, precision-driven arithmetic, and all the algorithms are implemented as part of a stand-alone software library, PRECISE. PRECISE is available for non-commercial use at `http://www.research.att.com/~krishnas/PRECISE`.

As compared to many earlier algorithms, our approach provides the following benefits:

- **Arbitrary size of input primitives:** We make no assumptions about the size of input primitives and their bit-lengths.
- **Arbitrary size of geometric predicates:** We make no assumptions on the size of geometric predicates and algebraic expressions.
- **Coupled representation:** Our numeric representation includes arbitrary precision plus an error term. As a result, we can also represent inputs with errors.
- **Iterative precision determination:** The computational framework determines precision by successive refinement, allowing us to reliably perform computations where explicit expression dags are not realizable.
- **Efficiency:** Our implementations work well in practice. Using PRECISE, we were able to compute the exact boundary representation of the intersection of two tori, which ultimately entailed isolating the roots of a degree-88 polynomial with coefficients of more than 4900 bits. In general, as compared to techniques using exact rational arithmetic throughout, we obtain between 1-2 orders of magnitude speedup in isolating roots of moderate degree with moderate bit lengths.

Organization: The rest of the paper is organized in the following manner. We give an overview of the algebraic queries in Section 2. We describe our numeric representation in Section 3 and show it can be used for efficiently and accurately computing the sign of all algebraic predicates. We present a novel root finding algorithm in Section 4 that can compute all the roots of a (possibly ill-conditioned) univariate polynomial up to any desired precision. We compare our techniques with other approaches in Section 5. In Section 6, we highlight PRECISE's performance on accurate computation of curve arrangements (Section *6.2.1*) and boundary evaluation of low degree solids (Section *6.2.2*). We conclude in Section 7.

2. BACKGROUND

A number of geometric algorithms depend on the evaluation of a few algebraic predicates. This has been well established through the work of Fortune and Van Wyk [20], Yap [42], Shewchuk [38], Burnikel et. al. [9, 10], and Devillers [16]. Many non-linear geometric problems such as boundary evaluation of boolean combination of curved solids, medial axis computation of polyhedra and algebraic curve arrangements, these predicates are, or depend on, the signs of univariate polynomials and determinants. The numeric entries of these expressions are often obtained as a result of performing symbolic elimination algorithms such as resultants. The bit sizes of these entries grow quickly as a function of the degrees and typically cannot be represented using 64-bit fixed precision arithmetic.

An example of how these geometric predicates arise can be seen by considering the boundary evaluation problem [27]. In this problem, one must evaluate the intersection between two parametric patches. The intersection curve can be expressed in the domain of one patch as the zero set of a bivariate polynomial. Finding points on the intersection curve (such as local maxima and minima) may involve finding the intersections of two such curves. There are multiple ways of finding the intersections of the curves, but one of the effective methods involves performing resultant calculations and then analyzing the roots of a series of univariate polynomials [28]. The accurate computation of the resultant and the roots of the resulting univariate polynomials are the dominant time consuming steps in the entire boundary generation process.

One commonly used algorithm for exact univariate real root finding is based on evaluating the signs of polynomials in a *Sturm sequence* [35], which is related to the polynomial remainder sequence that arises in GCD computations. Constructing the sequence involves generating polynomials that have high bit length coefficients, while evaluating the sequence involves finding the sign of these polynomials (i.e. computing algebraic predicates). The performance of the boundary evaluation algorithm (in terms of efficiency and accuracy) is strongly influenced by the performance of the algorithms for representing and computing the algebraic predicates.

3. NUMBER REPRESENTATION IN PRECISE

For the kind of geometric problems we are interested in, we would like a number system for reals that can:

- Handle all kinds of inputs (integer, rational, real) whose bit sizes are not bounded
- Provide adaptive precision arithmetic
- Guarantee the number of significant digits after a sequence of operations
- Perform reliably when computation cannot be represented in the form of a closed expression or when it is iterative
- Handle input data with error and provide mechanisms to specify the error.

This section gives some details about the number system we have designed and implemented in the PRECISE library, which has the properties above. Our approach is based on interval arithmetic using an extended precision floating-point data type. Because our floating point numbers can be arbitrarily long, we represent the interval not as a pair of floating point numbers, but as a pair (x, ϵ), where x is an arbitrary-precision floating point number indicating the center of the interval, and ϵ is a fixed precision number indicating the width. As a further (and significant) economy, the length of x is reduced whenever the interval becomes wide enough (as indicated by ϵ) to render meaningless the last machine word in the representation of x. Details are given in the following sections.

3.1 Floating Point Representation

Each floating point number (termed *real* in PRECISE) consists of a $(mantissa, exponent)$ pair, (m, e). For reasons described at the end of Section 3.2, we use a decimal system to represent our *real*s as opposed to a binary representation. Each number is represented conceptually as $(sign).d_1 d_2 \cdots d_n \times B^e$, where the decimal digits d_i form the mantissa, B is the base, and e is the exponent. For storage optimization, we actually use a larger value for the exponent base than 10, choosing the largest power of ten whose square will fit in a word of memory. For example, if a memory word is 32 bits long, $B = 10^4$. The extra space in a machine word is used in the multiplication algorithm. Thus, our new representation takes the form $(sign).m_1 m_2 \cdots m_n \times B^e$, where each m_i fits in one memory word. A floating-point number now requires a variable amount of memory, n units for the mantissa elements and one for the exponent, and some additional space to record the sign and the mantissa length. The leading mantissa element must be a nonzero block. The constant 0 can be identified by a zero value for the mantissa length, and no exponent or mantissa units need to be allocated.

Algorithms to perform the basic four arithmetic operations on such representations are described in Brent [6] and Knuth [29]. For each of these operations one may specify an upper bound on the number of result mantissa elements. Such a bound is clearly needed for the division operation, which can possibly result in a nonterminating result mantissa. Even though the other operations can be performed without this bound, a complicated sequence of these operations can quickly lead to very long mantissas and increasing operation costs. We overcome this problem in our system by maintaining a global variable called *precision* which can be changed at any time during the program. Whenever the value of *precision* is changed, all the new computations result in mantissas not exceeding this length.

3.2 Error Maintenance

In addition to the above representation, we also associate an error term ϵ with each floating point number. A number now has the representation

$$(\text{sign}).m_1 m_2 \cdots (m_n \pm \epsilon) \times B^e.$$

The error term ϵ holds as many decimal digits as one mantissa element and matches the last element m_n in its significance. By adding or subtracting ϵ from the last element in the mantissa we generate the endpoints of the interval in which the number must lie. When there is no error, $\epsilon = 0$. When an arithmetic operation is performed on two exact numbers, the result is initially computed exactly, and then truncated if necessary to the number of digits given by *precision*. If no truncation occurs, ϵ is still 0. Otherwise ϵ becomes 1, indicating that there is now a maximum error of one unit in the last mantissa decimal place.

Arithmetic operations on operands which are possibly both inexact (having a positive error) amount to interval arithmetic operations, and the rules governing these operations are well known [33, 34]. We initially compute the error and the mantissa of the result. If the error is now too large to fit in single memory word, the mantissa is shortened and ϵ becomes the maximum error in the last element of the new mantissa. Thus, as significance is lost, the lengths of the numbers represented is reduced. Consequently, the later operations, which are dependent on mantissa length, execute faster. If, at the end of the computation, the numerical significance is insufficient, the variable *precision* is increased, increasing the number of digits with which to start, and the entire computation is repeated (iterative revision).

There is another advantage of this interval-based representation. Most of the applications which require complicated numerical computations get their input through measured or derived data. The inputs often have errors associated with them. Adaptive precision computation libraries which do not represent errors are forced to assume that the input data are exact and proceed with the calculations. However, our system handles not only computational errors but representational errors also.

The advantage of the decimal system over binary lies mainly in the conversion of constants like 0.1 (here it is assumed that most input is given to us in a decimal notation) to the mantissa-exponent format. Only a few mantissa elements are required, and the conversion needs to be done only once. If the binary system were used, computation would be required for conversion to binary, with the number of mantissa elements equal to *precision*, which means the conversion has to be repeated for every increase in the *precision* parameter.

3.3 Sign Evaluation

In this section, we will briefly describe the technique we use to compute the signs of algebraic expressions using the numeric representation described above. In this representation, the sign of an expression is known when the expression evaluates to an interval not containing zero. In this section, we restrict our attention to rational expressions involving real numbers, which we shall call the *inputs* of the algebraic expression. We will assume that we can arbitrarily improve the precision with which the inputs are evaluated. For example, consider a polynomial expression with rational coefficients. To improve the precision of the evaluated expression at some rational point, we can evaluate these inputs (the coefficients and the point) to any precision we want.

Before we give our algorithm, we will state (without proof) a useful result we have obtained on the precision required to evaluate the rational number exactly in reduced form. Let r be a floating point approximation with the continued fraction expansion $\{f_0, f_1, \cdots, f_n\}$. The expansion is necessarily finite because r is rational. Consider the recurrence relations $p_k = f_k p_{k-1} + p_{k-2}$ and $q_k = f_k q_{k-1} + q_{k-2}, k = 0, 1, \cdots n$ $(p_{-2} = q_{-1} = 0, p_{-1} = q_{-2} = 1)$.

LEMMA 1. *Let r be a floating point approximation to the rational number $p/q < 1$ such that $|r - p/q| \leq \epsilon$, $|q| \leq B$ and $\epsilon < \frac{1}{2B^2}$. Let m be the largest index such that $q_m \leq B$. Then $p = p_m$ and $q = q_m$.*

The above lemma states that if we compute our rational approximation with enough precision, we can actually recover the rational number in reduced form. If only the sign needs to be computed, it is a trivial observation that $\epsilon < \frac{1}{2B}$ will suffice. In most of the expressions we evaluate, such as signs of determinants and signs of polynomials evaluated at specific values, it is easy to obtain the bound B specified in the lemma, e.g. via Hadamard's bound. This

result shows that our sign evaluation method will terminate with a correct answer.

Instead of performing all our computations at the level determined by the above lemma, we have adopted an optimistic strategy to evaluate the signs. We initially determine all the inputs of the expression at a small, implementation-specified precision and try to evaluate the sign. If the sign can be recovered, we are done. Otherwise, we iteratively increase the precision by a constant multiplicative factor and redo the entire computation as before until the sign can be determined unambiguously. If the actual result is zero, the error in the result will eventually fall below the ϵ (as specified in the above lemma) limit and we can stop the algorithm. It must be observed that this strategy works only under the assumption that the error in the inputs of the expressions can be refined arbitrarily. If not, the algorithm will try to determine the sign with the given precision of inputs, but it cannot guarantee an answer in all cases. We do not have any tight bounds on the precision required to reliably compute the signs of algebraic numbers.

4. UNIVARIATE ROOT FINDING WITH GUARANTEED ERROR BOUNDS

In this section, we present a root finding algorithm based on our representation. It computes all the roots of a polynomial and guarantees the number of significant digits in each root. The algorithm can handle ill-conditioned polynomials whose roots are usually clustered together. We use an iterative scheme that starts with an initial approximation of all the roots, refines them and updates the error bound. The basis for our algorithm is the Durand-Kerner method [18, 17]. We will present some new results based on Rouche's theorem to obtain good initial approximations for the roots.

4.1 Basic Notation and Algorithm Overview

We begin by describing the Durand-Kerner method briefly. Consider a monic polynomial $f(x)$ given by

$$\begin{aligned} f(x) &= x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\ &= \prod_{i=1}^{n}(x - r_i) \end{aligned} \tag{1}$$

The Durand-Kerner method is an iterative root-finding method analogous to Newton's method. Instead of beginning with a single estimate, it begins with a vector of estimates $(x_1^{(0)}, \ldots, x_n^{(0)})$ and applies the following recursion:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{f(x_i^{(k)})}{\prod_{j \neq i}(x_i^{(k)} - x_j^{(k)})} \tag{2}$$

There are a number of variants of this method proposed. There is an improved scheme given by Ehrlich [19] as follows:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{1}{\frac{f'(x_i^{(k)})}{f(x_i^{(k)})} - \Sigma_{j \neq i}\frac{1}{(x_i^{(k)} - x_j^{(k)})}} \tag{3}$$

This is the basis of the Durand-Kerner scheme for simultaneous polynomial root evaluation. The convergence behavior of this method has been well studied [2]. Alefeld and Herzberger [2] showed that the iteration scheme (eq. (2)) exhibits local quadratic convergence. We show that the improved iteration scheme (eq. (3)) exhibits local cubic convergence under certain conditions [30]. Little is known about the global convergence of this method. It has been shown that the Durand-Kerner scheme is globally convergent outside a set of measure zero for quadratic and cubic polynomials [21]. Numerical experiments, however, show that even for polynomials with multiple roots the method converges for almost all initial values. The conjecture for global convergence [21] is still unsolved. Our algorithm uses the Durand-Kerner approach as follows.

1. Choose initial approximations $x_i^{(0)}$, $i = 1, \cdots n$.
2. Execute one iteration of the Durand-Kerner scheme for all i.
3. If new iterates do not improve the previous ones, compute error bounds on the approximations.
4. If error bounds are not satisfactory, increase computation precision and goto step 1.
5. Report computed root approximations, their multiplicities and halt.

4.2 Choice of Initial Approximations

One of the critical steps for this iteration scheme to work is the choice of the initial approximations to the roots of the original polynomial. In this section, we will state the basic results which determine our choice of initial approximations. The proofs are omitted here for the sake of brevity. We refer the reader to [30] for complete details.

We now state Rouche's theorem [37, 24] from classical complex analysis. This forms the basis of our results.

THEOREM 1. *If $P(x)$ and $Q(x)$ are analytic interior to a simple closed Jordan curve C, continuous and non-vanishing on C, and $|P(x)| < |Q(x)|, x \in C$, then the function $F(x) = P(x) + Q(x)$ have the same number of zeros (counted with multiplicities) interior to C as $Q(x)$.*

We now recall a fundamental result of Cauchy [12].

THEOREM 2. *All the zeros of the polynomial $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, lie in the circle $|x| \leq r$, where r is the (only) positive real root of the equation*

$$-|a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0|$$

We are now ready to state our result, which is a generalization of Cauchy's theorem.

THEOREM 3. *Given a polynomial $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_k x^k + \cdots + a_1 x + a_0$, if the associated polynomial $f_k(x), k \neq 0, n$,*

$$|a_n|x^n + \cdots + |a_{k+1}|x^{k+1} - |a_k|x^k + \cdots + |a_0|$$

has two positive real roots r_k and R_k, $r_k < R_k$, then $f(x)$ has exactly k zeros inside or on the circle $|x| \leq r_k$, and no zeros in the annulus $r_k < |x| < R_k$.

Cauchy's theorem appeared in 1829. Given that the generalization of his theorem is not too difficult, we would not be surprised if our generalization has been proven before. However, we have not been able to find any reference to this theorem in the literature.

THEOREM 4. *Let the associated polynomial $f_k(x), k \neq 0, n$ of the original polynomial $f(x)$ be defined as above. Then $f_k(x)$ can have either zero or two positive real roots.*

The basic use of the above theorems is to provide insight into the positions where roots could occur. Most of the previous root finding schemes use Cauchy's theorem to generate the bound on all the roots and then try to converge to the roots. The problem with this method is that if the roots of the polynomial are clustered with very large variation in the size of these roots (as is often the case with ill-conditioned polynomials), the iteration scheme will either fail or converge very slowly.

So, using theorem 3, let the values of k for which $f_k(x)$ has 2 positive real roots be $k_1 < k_2 < \cdots < k_{m-1}$. It is easy to show that $f_0(x)$ and $f_n(x)$ each have exactly one positive real root. Adding 0 and n to each end of the above list gives us the sequence $0 = k_0 < k_1 < k_2 < \cdots < k_{m-1} < k_m = n$. Let us call this list *k_list*. For these indices, let the corresponding roots $R_{k_0} < r_{k_1} < R_{k_1} < r_{k_2} < R_{k_2} < \cdots < r_{k_{m-1}} < R_{k_{m-1}} < r_{k_m}$ (termed *k_root_list*. Also, from theorem 3, we know that there are exactly $(k_{i+1} - k_i)$ roots in the annulus $R_{k_i} < |x| < r_{k_{i+1}}$, $0 \leq i \leq (m-1)$. We now give the placement of the initial approximations in our algorithm.

$$x_{k_j+l}^{(0)} = r_{k_{j+1}} e^{i\left(\frac{2\pi l}{(k_{j+1}-k_j)} + \phi\right)}, 0 \leq j < m, 1 \leq l \leq (k_{j+1} - k_j)$$

where $i = \sqrt{-1}$ and ϕ is an arbitrary angle chosen so as to avoid symmetry in the arrangement of estimated roots. In simple terms, we are just placing each set of $(k_{i+1} - k_i)$ approximations uniformly on a circle with a random initial offset. The above formula is very similar to the original form it appeared in Aberth's [1] paper, except that he had the placement for all the n roots in a single large circle.

We observe that since r_k and R_k need to be evaluated only for initial root placement, it is sufficient to compute them approximately. We refer to them as r'_k and R'_k respectively. We now describe a very efficient method to evaluate r'_k and R'_k . We start with the method for estimating r'_n (root of $f_n(x)$). We know that $f_n(0) > 0$ and $f_n(\infty) < 0$. Our method starts by finding a small positive value δ such that $f_n(\delta) > 0$. We double δ repeatedly until the function value becomes negative. It is possible to do some finer iterations between the last two δ values to generate a more precise upper bound for r'_n. A similar strategy can be applied to find R'_0 (except that we approach the root from the opposite side). In order to obtain r'_k and R'_k, we observe that $f_k(x) < 0$, $r_k \leq |x| < R_k$. That is,

$$\begin{aligned} |a_k|x^k &> \sum_{i \neq k} |a_i|x^i \\ &> \sum_{i<k} |a_i|x^i \end{aligned} \tag{4}$$

Also,

$$\begin{aligned} |a_k|x^k &> \sum_{i \neq k} |a_i|x^i \\ &> \sum_{i>k} |a_i|x^i \\ |a_k| &> \sum_{i=1}^{n-k} |a_{i+k}|x^i \end{aligned} \tag{5}$$

Equation 4 is used to evaluate r'_k using the method described above for finding r'_n. Equation 5 is used to evaluate R'_k using the method for R'_0. We perform this computation for $k = 1, 2, \cdots, n$. If the computed bounds r'_k and R'_k are such that $r'_k > R'_k$, we ignore this value of k from the sequence. This placement algorithm works very well on polynomials which exhibit root clustering and clusters which are far apart in the complex plane.

We shall briefly show the benefit of the above result with a simple numerical example. Consider the polynomial

$$f(x) = x^3 - 72.1 * x^2 + 148.1 * x - 77$$

whose roots are (1.0,1.1,70.0). By applying Cauchy's theorem, we find that the size of all the roots is bounded by 74.11. Applying our result to the above polynomial gives us a *k_list* $\{0, 2, 3\}$ with the corresponding *k_root_list* $\{0.43, 2.56, 69.97, 74.11\}$. This says that there are two roots whose size lies in the interval $[0.43, 2.56]$ and one root in $[69.97, 74.11]$, which is correct. By placing the initial approximations in the appropriate circles, the convergence is improved substantially.

4.3 Algorithm with Multiprecision Arithmetic

Now that all the initial approximations are found, we are ready to perform each step of the iteration. As consecutive iterates are found, we need to compute the absolute error in each approximation. For this, we make use of a result from Smith [40].

THEOREM 5. *Let $x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)}$ be distinct and let $\sigma_j = f(x_j^{(k)})/g'(x_j^{(k)})$ for $j = 1, \cdots, n$ ($g(x) = \prod_{i=1}^{n}(x - x_i^{(k)})$). Define*

$$\Gamma_i : |x - x_i^{(k)}| \leq n|\sigma_i|, i = 1, \cdots, n$$

Then the union of the circles Γ_i contains all the roots of $f(x)$. Any connected component of this consisting of m circles contains exactly m roots of $f(x)$.

We use the above result to compute the absolute error in each iterate. The root finding algorithm proceeds in the following steps:

- If all the circles are isolated, we have achieved root isolation and if the radius of these circles is smaller than the precision limit, we are done.
- However, if there are clustered roots, it is possible that some of the circles are connected. In this case we compute the worst case error $\epsilon_i = \max(|x_i^{(k)} - x_j^{(k)}| + n|\sigma_j|)$, where j ranges over the set of indices for which $x_j^{(k)}$ is part of the same connected component as $x_i^{(k)}$.
  - If ϵ_i is smaller than the precision desired, we report a multiple root.
  - Otherwise, we redistribute these approximations on a single circle with center at the centroid of the iterates and radius equal to $\max_i(\epsilon_i + n|\sigma_i|)$, where i belongs to the iterate indices of the same connected component of the circles. The redistribution places the approximations uniformly by angle as before. Only the approximations in these clusters are refined in future steps. We found that this method accelerates the convergence process in case there is a higher multiplicity root near our iterates.

We perform all the computations using PRECISE reals. Since all the results of our computations have guaranteed error bounds, we are assured of root separation if the circles determined by the error bounds are not connected.

4.4 Performance on Test Polynomials

In order to observe the performance of our algorithm, we used a well established benchmark of polynomials collected by the PoSSo (Polynomial System Solving) project in Europe (`http://www-sop.inria.fr/saga/POL`). Each of these polynomial classes are known to give problems (due to poor conditioning or root clustering) to other root finding algorithms. We ran these test cases on two other established root finding software systems—Maple (from University of Waterloo) and MuPAD (from University of Paderborn). We first list each of the polynomial classes.

- **Poly1:** $\Sigma_{i=0}^{n} \frac{x^i}{i!}, n = 50$. The roots of this polynomial are located on a curve.
- **Poly2:** $(x - 3c^2)^2 + icx^7, 0 < c \ll 1, c = 10^{-20}$. Degree 7 polynomial , real and imaginary parts of the coefficients are rational. The polynomial has two simple complex roots with extremely small real separation (113 common digits) and imaginary parts close to zero (order of 10^{-149}). The remaining roots are well separated.
- **Poly3:** $(c^2x^2 - 3)^2 + c^2x^9, c = 10^{20}$. Degree 9 polynomial, integer coefficients. The polynomial has two extremely close real roots (70 common digits) plus a complex pair with extremely small imaginary part (less than 10^{-90}). The remaining roots are well separated. The condition number of the above clustered roots is greater than 10^{40}.
- **Poly4:** $x^{20} + cx^{14} + x^5 + 1$, where c is a number with a fairly large modulus. In our case, we set $c = 10^{12}$. The main problems with this polynomial is that the intermediate values are likely to cause overflow if multiprecision representation is not possible.
- **Poly5:** $\prod_{i=1}^{n}(x - i), n = 40$. This is the classic Wilkinson polynomial of degree 40. Here the coefficients are very large and the conditioning of the roots is very high (ranging from 10^2 to 10^{14} for $n = 20$).
- **Poly6:** $(0.01x^{10} + (x - 10)^2) \prod_{i=1}^{20}(x - i)$. This is a modified case of the Wilkinson polynomial with both conditioning problems and root clustering.
- **Poly7:** $\prod_{i=1}^{20}(x - i)(x - 20)^2$. This polynomial has the ill-conditioning problems of the Wilkinson polynomial along with root multiplicities.
- **Poly8:** $x^{14} + 2cx^{11} + c^2x^8 + 4x^7 - 4cx^4 + 4, c = 10^{24}$. This is a case where of the 14 roots, 8 roots are clustered with moduli around 10^{-6} and 6 roots clustered with moduli around 10^8. This is a good case for our result on initial choice of approximation.
- **Poly9:** $x^n - a, n = 50, a = 1$. Sparse polynomial. The polynomial has roots uniformly distributed along a circle, for a=1 they are the n-th roots of the unity. Even though the roots come close together as n grows, they are numerically well conditioned. Their condition number is at most $2/n$.

Figure 1 tabulates the running time of our algorithm compared to that of Maple and MuPAD on each of the above cases. Since Maple and MuPAD do not have the facility of guaranteeing the number of significant digits of their output, we ran their software with the same number of digits of precision as the maximum used by PRECISE for that polynomial. All the times are in seconds and measured on an SGI Origin 400 MHz R12000 processor running Irix6.5. We are not aware of the algorithm that is implemented for polynomial root solving in Maple and MuPAD. It can be observed that PRECISE performs very competitively with the other two software even though it performs extra work to guarantee significant digits. We were recently informed by one of the reviewer's of the existence of another algorithm and implementation for univariate root solving based on Aberth's method [1] called MPSolve provided by Bini et. al. [5]. While we were not able to test its performance on our machines, the timings provided in their paper seem to be an order of magnitude faster than our algorithm.

Case	Root Precision	MuPAD	Maple	PRECISE
Poly1	10	78.77	4.79	13.17
Poly2	120	5.15	41.54	13.06
Poly3	80	3.32	8.539	6.32
Poly4	30	4.569	0.792	0.82
Poly5	30	32.21	36.31	5.67
Poly6	30	14.01	34.31	0.65
Poly7	30	6.517	13.53	1.23
Poly8	30	12.21	18.717	5.26
Poly9	30	71.215	1.26	1.44

Table 1: Univariate root finding algorithm applied to nine polynomials from the PoSSo benchmark suite. The second column indicates the number of significant digits of the roots PRECISE was asked to compute. The last three columns indicate running time taken by Maple, MuPAD and PRECISE. Times are in seconds and measured on an SGI Origin 400 MHz R12000 processor running Irix6.5.

4.5 Computing Real Roots of a Polynomial

The previous section described our general algorithm for computing all the roots of a polynomial and all the arithmetic is done in complex space. However, in most geometric applications we are interested only in the real roots. Bairstow's method [41] is a well-known algorithm in numerical analysis to compute the real roots using only real arithmetic. The main idea of this method is that instead of trying to factor out linear factors of a polynomial with real coefficients, it factors out quadratic factors. The coefficients of these factors are always real since the roots occur in conjugate pairs. Thus Bairstow's method avoids complex arithmetic altogether.

Consider a polynomial with real coefficients $f(x)$ as in equation 1. Let (A, B) and (A_1, B_1) be the coefficients of the linear remainder such that

$$f(x) = f_1(x)(x^2 - rx - q) + Ax + B$$
$$f_1(x) = f_2(x)(x^2 - rx - q) + A_1x + B_1$$

Then Bairstow's method [41] simply iterates to compute refinements on the coefficients of the quadratic factor (r, q).

$$\begin{pmatrix} r^{k+1} \\ q^{k+1} \end{pmatrix} = \begin{pmatrix} r^k \\ q^k \end{pmatrix} - \begin{pmatrix} A_1r^k + B_1 & A_1 \\ A_1q^k & B_1 \end{pmatrix}^{-1} \begin{pmatrix} A \\ B \end{pmatrix}$$

Essentially, we have extended this sequential version of Bairstow's method to one which iterates over all the (r_i, q_i) pairs simultaneously based on an observation in the paper by Handscomb [23]. Details of this extension are omitted in this paper. Our implementation of this algorithm under PRECISE is about 20% faster than the previous algorithm for most of the polynomials we tested.

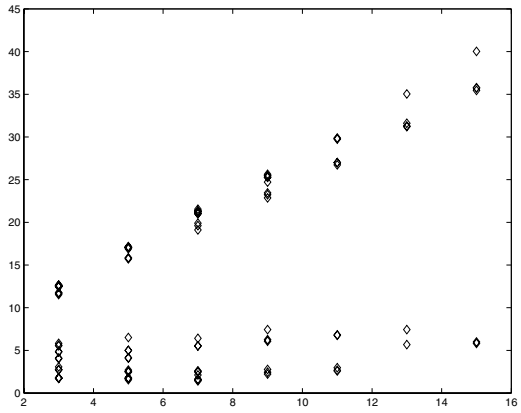

Figure 1: Determinant sign speedup. The speedup factor for PRECISE, for various matrices. The horizontal axis gives the order of the matrix, while the vertical axis gives the speedup factor in comparison to an exact modular arithmetic algorithm. The set of matrices includes randomly generated matrices and others arising in geometric problems. The matrices are all non-singular.

5. ALGORITHM COMPARISON

In this section, we compare our algorithm with other algorithms for efficient and accurate computations. The underlying numerical representation is based on automatic forward error propagation. As such it is closely related to three approaches: interval arithmetic, expression tree methods, and expression compilers.

Interval arithmetic can be directly applied as a filter for any expression. The LEDA `floatf` type is an example [32]. PRECISE extends this idea by supporting a repetition of the calculation at higher precision. The advantage of higher precision is that PRECISE can accept input of any precision and evaluate predicates exactly.

Expression tree methods, such as those implemented in LEDA's `real` data type [32] and the CORE library's `Expr` data type [25], are a further generalization. In this model, associated to each quantity is not only an error bound but an explicit formula for recomputing the quantity at a higher precision. The formula is in the form of a tree or directed acyclic graph. These methods also allows for *precision-driven computation*: the propagation of desired error bounds from the top of the tree to the leaves [43]. This avoids recomputing the expression at an intermediate precision that turns out to be insufficient. Expression tree methods appear to be practical only for small- to moderate-length expressions. They have yet to be applied to the problem of the sign of a large determinant, and are clearly impractical for iterative algorithms like Durand-Kerner for computing roots of high degree polynomial systems.

Expression compilers, like LN [20], cut down on the run-time overhead of expression tree methods by deriving error bounds from the structure of the expression before its variables are specialized. They likewise have limited application to large problems, high-precision inputs, and iterative algorithms.

Many more authors have focused on the problem of the sign of the determinant of a small matrix [13, 8, 20, 38, 3]. Little of that recent work applies to computing determinants of large matrices. Some of these approaches also limit the input precision, which is difficult in the context of dealing with arbitrary degree algebraic numbers. The modular-arithmetic approach used in LiDIA [22] and extended by [7] applies to matrices with integer entries only. Within this context, the iterative reconstruction algorithms are interesting in that their running times are determined by the magnitude of the determinant, which is not necessarily related to the precision required by a forward-error-based approximate method. The floating-point filter for sign-of-determinant based on the SVD, presented in [28], works well for large matrices, and correctly computes the sign of a well-conditioned matrix. The SVD filter is distinguished from other floating-point filters in that it is based on backward error analysis rather than forward.

6. PERFORMANCE AND APPLICATIONS

In this section, we highlight PRECISE's performance when applied to the evaluation of determinant signs, arrangements of curves, boundary representations, and medial axes. All times presented are in CPU seconds on a 300 MHz R12000 MIPS processor.

6.1 Determinant Sign

The PRECISE library has been applied to computing the sign of the determinant of large rational matrices. The algorithm used is Gaussian elimination with partial pivoting using the *real* arithmetic of PRECISE with progressively higher precision.

When applied to nonsingular matrices, we have found that this method is faster than exact methods that use modular arithmetic, as shown in Figure 1. Culver [14] proposes a filter for computing exact determinant signs. Using singular value decomposition (SVD), it is possible to estimate whether a matrix is singular, and even to determine the sign of the determinant directly if the matrix is well conditioned. If it is ill-conditioned but is likely to be singular, modular arithmetic is used. If it is ill-conditioned and unlikely to be singular, the PRECISE determinant sign computation is performed.

6.2 The MAPC Library

The MAPC library [28] provides facilities for exact manipulation of algebraic points and curves. The PRECISE library has been incorporated into MAPC as a filter to accelerate the exact computation. A primitive operation for MAPC is the determination of PERM(f, x), the number of *sign permanencies* in the *Sturm sequence* of f evaluated at x. Up to a sign factor, the Sturm sequence of a polynomial f is the remainder sequence that arises in computing the GCD of a f and its first derivative. To compute the number of permanencies at a point x, one evaluates each polynomial of the sequence at x and counts the number of times the sign remains same between one polynomial and the next. The computation of permanencies is useful because $PERM(f, x_2) - PERM(f, x_1)$ yields the number of distinct real roots of f between x_1 and x_2. The MAPC library has used exact arithmetic capabilities based on LiDIA [22] to accurately compute the Sturm sequences. However, for high degree polynomials, exact computation of Sturm sequences can be prohibitively slow because of the very large growth in the bit lengths of the coefficients. In the worst case, the bit length can double with every numeric division, and the number of accumulated divisions in a Sturm sequence computation can be of order n^2 in the degree of the original polynomial. Some solid modeling computations can require a Sturm sequence computation for polynomials of degree > 80.

To resolve this problem, we compute the Sturm sequence and permanencies using PRECISE's naive precision-driven computational model. Since we are only interested in the sign of the polynomials, the computation usually has enough precision to determine the signs exactly. Occasionally, it can happen that the result has an interval containing zero. When this occurs, the calculation of the Sturm sequence and the number of permanencies is repeated at

Case	1	2	3	4
Number of Curves	3	3	6	7
Coefficient Bit size	25	22	25	62
Number of Faces	9	11	31	63
Total time without PRECISE	5.2	9.0	62.8	122.8
Total time with PRECISE	1.8	4.0	11.0	9.3
Time computing resultant	1.2	3.1	8.1	4.5
Time generating Sturm sequence without PRECISE	3.6	5.8	54.2	117.1
Time counting permanencies without PRECISE	0.06	0.03	0.03	0.09
Time generating Sturm sequence in PRECISE	0.3	0.3	1.1	2.1
Time counting permanencies in PRECISE	0.1	0.2	0.9	1.2

Table 2: The table shows the maximum bit length needed to express the coefficients of the curves shown in Figure 2, the number of faces generated by the arrangement, the time taken using the original (exact rational based) code, and the time taken using PRECISE. The curves have maximum degree 4. Times are in seconds on a 300 MHz R12000 MIPS processor.

a higher precision. After a fixed number of attempts at increasing precision, a call is made to the routine using exact rationals. Currently this is done if the value cannot be distinguished from zero with 40 decimal digits of precision. In practice this occurs very rarely. Thus the PRECISE computation serves as a filter.

6.2.1 Curve Arrangements

We have also applied PRECISE to compute an arrangement of curves. Given a number of algebraic curves in a plane, the goal of the curve arrangement algorithm is to compute the subregions (called the "faces") of the overall region which are connected without any curve passing through them. Each face is defined by piecewise algebraic curves that enclose its boundary. All points in one face will have the same sign with respect to all of the given polynomials. The output of the curve arrangement algorithm is the explicit topological description of each cell.

Table 2 shows a comparison of the time taken for some arrangement computations for curves shown in Figure 2 with MAPC implementation [28] and the new PRECISE library for Sturm sequence computations. It is clear from these examples that PRECISE can provide significant speedups for such algebraic computations.

6.2.2 Boundary Generation

A motivating application for the MAPC library is the boundary evaluation of (low degree) algebraic solids. MAPC is a core library in the ESOLID system [27] performing such boundary evaluations. In general, the problem of determining a boundary representation for a CSG model is a challenging one that raises problems of robustness. These problems are exacerbated when the CSG primitives have curved boundaries. To alleviate the robustness problems, the ESOLID system performs performs all geometric tests exactly, using layered filters to make the exact computation more efficient.

We have tested ESOLID on portions of a real-world model, the Bradley Fighting Vehicle provided courtesy of the Army Research Laboratory. Some example output B-reps are shown in Figure 3, with comparative timings given in Table 3.

Table 3 shows the performance improvement when PRECISE is applied to the test examples. PRECISE affects the performance of only the Sturm sequence portion of ESOLID. As seen in the table, PRECISE can dramatically lower the running time of the most Sturm sequence intensive examples. In the M16 example (Figure 3(c)) the Sturm code is made almost two orders of magnitude faster. This makes the overall computation more than an order of magnitude faster. PRECISE incurs a certain amount of overhead, which is the reason that several of the low-Sturm time examples actually become slower when run with PRECISE.

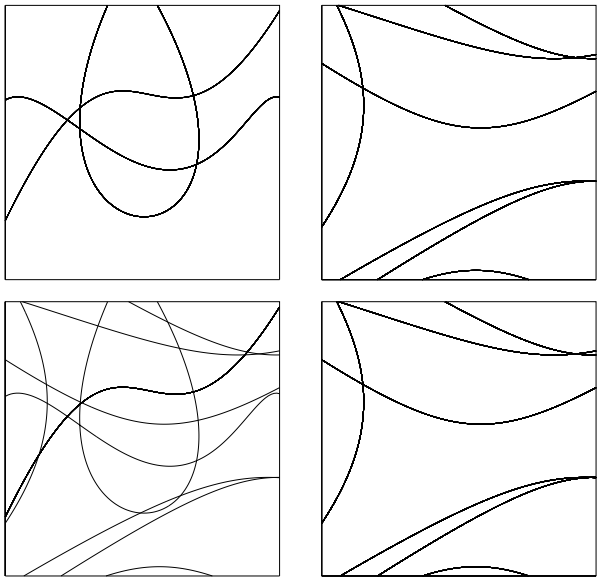

Figure 2: Arrangement of planar algebraic curves. The figures show the curves which partition the region into faces. The application finds all subregions, the segments of curves bounding each subregion, and the connectivity between subregions. Cases 1 and 2 are on the top row, 3 and 4 on the bottom.

6.3 Medial Axis Computation

Computing the medial axis of a polyhedron is a challenging problem because it inherently requires analysis of intersecting curved surfaces. Culver et al. [15] have implemented an exact algorithm for medial axis evaluation that relies both on the MAPC library and on the sign of determinant filter described in Section 6.1, both of which incorporate PRECISE. The program has been used to compute the medial axis of complicated polyhedra with as many as 250 faces [14]. An example of the output is given in Figure 4.

7. CONCLUSIONS

We have designed and implemented PRECISE, a floating point number type that maintains an error bound and can operate to arbitrary precision. We have demonstrated its utility in the following settings:

- **Polynomial root solving:** We have presented a new polynomial root-finding algorithm using number representation in PRECISE. It computes all the roots with guarantees on the number of significant digits in each root. The algorithm can handle ill-conditioned polynomials which are traditionally known to cause problems for most polynomial solvers. The efficiency of the method also compares favorably with some of the most widely used solvers like Maple.
- **Exact geometric computation.** We have also used PRECISE as part of a multistage filter to accelerate exact geomet-

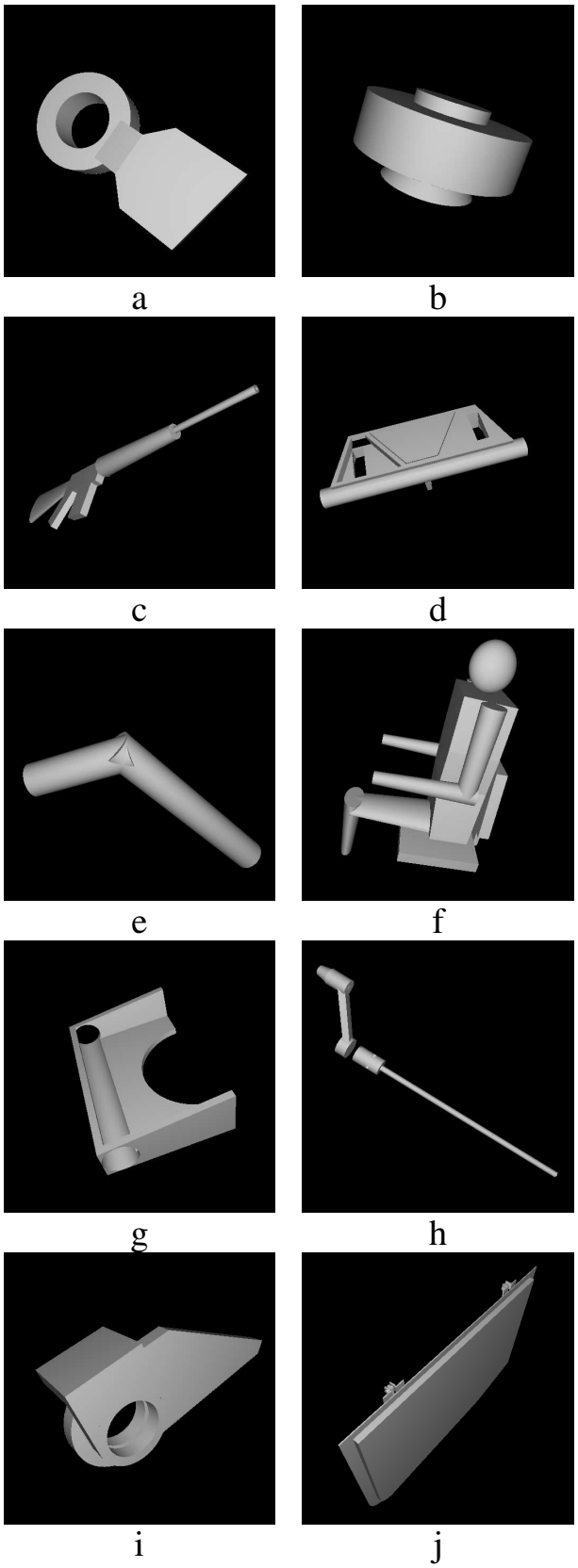

Figure 3: BRL-CAD examples.

Example Number	without PRECISE Total Time	without PRECISE Sturm Time	with PRECISE Total Time	with PRECISE Sturm Time
a	10.23	0.51	10.95	1.62
b	12.57	0.24	12.69	1.44
c	633.42	597.33	42.99	6.93
d	132.48	4.95	137.64	11.55
e	250.74	190.62	73.86	15.36
f	26.37	1.29	28.14	3.63
g	63.15	8.34	61.26	6.36
h	213.72	116.88	105.99	9.90
i	58.92	3.15	63.48	8.55
j	54.78	2.07	58.44	6.66

Table 3: Timings for the example from Figure 3, with and without the incorporation of the PRECISE library. PRECISE is used to improve the efficiency of Sturm sequence calculations. The total time and the time spent in Sturm computations is shown.

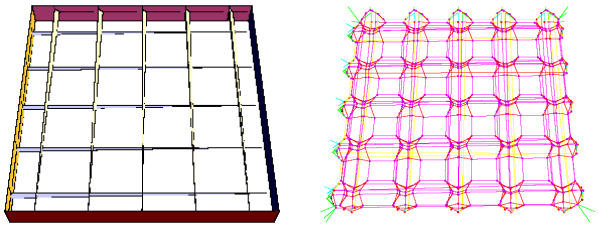

Figure 4: The "iron maiden pizza box" and a schematic of its medial axis. The top and bottom of the box are removed to show the spikes inside. This computation took 23 minutes.

ric calculations. Using PRECISE, we have seen a speedup of as much as an order of magnitude in computing curve arrangements and in boundary evaluation, as compared to the exact computation without the PRECISE-based filter. There is, however, a cost in the form of a slowdown when low-degree polynomials are involved.

Arbitrary precision interval arithmetic is a useful computational framework when high precision is needed in the result, and methods for determining the needed initial precision are unavailable. In such cases, the precision can be determined by iterative revision. These problems arise in practice, and we have applied the PRECISE library to their solution. In the future, we hope to investigate other areas where PRECISE can be applied.

8. REFERENCES

[1] O. Aberth. Iteration methods for finding all zeros of a polynomial simultaneously. *Mathematics of Computation*, 27(122):339–344, 1973.

[2] G. Alefeld and J. Herzberger. On the convergence speed of some algorithms for the simultaneous approximation of polynomial roots. *SIAM Journal on Numerical Analysis*, 11:237–243, 1974.

[3] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[4] David H. Bailey. A portable high performance multiprecision package. Technical Report RNR-90-022, RNR, 1993.

[5] D. Bini. Numerical computation of polynomial zeros by means of aberth's method. *Numerical Mathematics*, 13:179–200, 1996.

[6] R. Brent. A fortran multiple precision arithmetic package. *ACM Transactions on Mathematical Software*, 4:57–70, 1978.

[7] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[8] Hervé Brönnimann and Mariette Yvinec. A complete analysis of Clarkson's algorithm for safe determinant evaluation. Research Report 3051, INRIA, 1996.

[9] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 702–709, 1997.

[10] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 341–350, 1999.

[11] J. Canny. A new algebraic method for robot motion planning and real geometry. In *Proc. 28th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 39–48, 1987.

[12] A. L. Cauchy. Exercises de mathematique. *Oeuvres*, 9(2):122, 1829.

[13] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, October 1992.

[14] T. Culver. *Accurate Computation of the Medial Axis of a Polyhedron*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

[15] Tim Culver, John Keyser, and Dinesh Manocha. Accurate computation of the medial axis of a polyhedron. In *Proc. Symposium on Solid Modeling and Applications*, pages 179–190, 1999.

[16] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Proc. of ACM Symposium on Computational Geometry*, pages 139–148, 2000.

[17] K. Dochev. *Physical and Mathematical Journal of the Bulgarian Academy of Sciences*, 5:136–139, 1962.

[18] E. Durand. Solutions numeriques des equations algebriques. *Tome I, Masson, Paris*, 1960.

[19] L. W. Ehrlich. A modified newton method for polynomials. *Communications of ACM*, 10(2):107–108, 1967.

[20] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, May 1993.

[21] M. W. Green, A. J. Korsak, and M. C. Pease. Simultaneous iteration towards all roots of a complex polynomial. *SIAM Review*, 18:501–502, 1976.

[22] LiDIA Group. A library for computational number theory. Technical report, TH Darmstadt, Fachbereich Informatik, Institut fur Theoretische Informatik, 1997.

[23] D. C. Handscomb. Computation of the latent roots of a hessenberg matrix by bairstow's method. *Computer Journal*, 5:139–141, 1962.

[24] P. Henrici. *Applied and Computational Complex Analysis, Vol. 1*. John Wiley & Sons, New York, 1974.

[25] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 351–359, 1999.

[26] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, January 1991.

[27] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate b-rep generation of low degree sculptured solids using exact arithmetic I: Representations and II: Computation. *Computer Aided Geometric Design*, 16(9):841–882, October 1999.

[28] John Keyser, Tim Culver, Dinesh Manocha, and Shankar Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. In *Proc. 15th Annual ACM Symposium on Computational Geometry*, pages 360–369, 1999.

[29] D. Knuth. *The Art of Compter Programming, Vol. 2*. Addison-Wesley, Reading, MA, 1981.

[30] S. Krishnan, M. Foskey, J. Keyser, T. Culver, and D. Manocha. Precise: Efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. Technical Report TD-4R5SBW, AT&T Labs - Research, 2000.

[31] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *Proceedings of the Symposium on Discrete Algorithms*, 2001.

[32] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[33] R. E. Moore. The automatic analysis and control of error in digital computation based on the use of interval numbers. *Error in Digital Computation*, 1, 1965.

[34] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[35] P. Pedersen. Multivariate sturm theory. In *Proceedings of AAECC*, pages 318–332. Springer-Verlag, 1991.

[36] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proc. 10th Symp. on coputer arithmetic*, pages 132–143, 1991.

[37] E. Rouche. Memoire sur la serie de lagrange. *Journal of Ecole Polytech*, 22:217–218, 1862.

[38] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[39] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[40] B. T. Smith. Error bounds for the zeros of a polynomial based upon gerschgorin's theorem. *Journal of ACM*, 17:661–674, 1970.

[41] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1993.

[42] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[43] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. 1995.