Efficient and Reliable Computation with Algebraic Numbers for Geometric Algorithms

J. Keyser  S. Krishnan†  D. Manocha  T. Culver
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{keyser,krishnas,dm,culver}@cs.unc.edu
March 24, 1998

Abstract: Many geometric algorithms involve dealing with numeric data corresponding to high degree algebraic numbers. They come up in computing generalized Voronoi diagrams of lines and planes, medial axis of a polyhedron and geometric computation on non-linear primitives described using algebraic functions. Earlier algorithms dealing with algebraic numbers either use fixed precision arithmetic or techniques from symbolic computation. While the former can be inaccurate, the latter is too slow in practice. We present efficient representations and algorithms for reliable computations with algebraic numbers. We use these representations to efficiently perform geometric queries like inside/outside tests, which-side or orientation tests. The overall approach combines different techniques from symbolic computation based on exact arithmetic with floating point arithmetic. We demonstrate its applications to efficient and reliable computation of curve and surface intersections. In practice, it is about one order of magnitude faster as compared to earlier implementations that produce reliable results.

*Supported in part by a Sloan fellowship, ARO Contract D-34982-MA, NSF grant CCR-9319957, NSF grant CCR-9625217 and ONR Young Investigator Award.
†Currently at AT & T Research Labs
1 Introduction

Most geometric algorithms are based on the “Real RAM” model of computation, which assumes unit cost operations on real numbers. In practice, most computer programs replace the exact real arithmetic of this model by fixed precision arithmetic. However, geometric algorithms are known to be highly sensitive to numerical inaccuracies produced due to fixed precision arithmetic.

Many approaches have been proposed in the literature to handle this problem. These include the design of geometric algorithms such that robust implementations can be obtained using only the fixed precision hardware [DSB92, For95, Hof89, Mil89, Sug89]. However, designing such algorithms is quite involved and has been restricted to only a few problems. A second approach advocates the use of exact real arithmetic. However, a naive implementation of exact arithmetic can be quite slow and a number of techniques have been proposed in the literature to speed it up. One proposed solution has been to compute certain predicates exactly [BKM+95, EM90, FV93, Yap97]. This may involve computing a few algebraic expressions to a high enough precision to answer queries [FV93, Yap97] or using an algorithm that performs a specific test exactly. In the latter category, most of the work has been on computing the sign of a multivariate polynomial with integer coefficients or the sign of a determinant of a matrix with integer entries.

In this paper, we focus on efficient and accurate computations with high degree algebraic numbers for geometric applications. Such computations frequently come up in many geometric problems. The set of problems includes:

- Computing generalized Voronoi diagrams of configurations of lines and planes.
- Computing the medial axis transform of a polyhedron.
- Geometric computations on non-linear primitives described using algebraic functions. The primitives include rational spline curves and surfaces, NURBS, algebraic sets and their boolean combinations. The set of geometric algorithms include computing convex hulls, intersections, boundary evaluation of a primitive defined using CSG operations, etc.
It is rather difficult to design robust algorithms based on fixed precision arithmetic or iterative numerical methods for such problems [Hof89, KKM97]. An accurate algorithm involves computing an exact representation of geometric entities (e.g. vertices, curves or surfaces) and performing geometric queries (e.g. inside/outside tests, which-side or orientation tests) with these representations. In many cases, such tests cannot be reduced to computing sign of a single algebraic predicate (such as the sign of a determinant).

A number of algorithms have been proposed in the symbolic computation literature to represent and compute high degree algebraic numbers [Buc89, Can88, Col75, CKL89, Reg95]. Many of them have been implemented, however, their performance is too slow for most geometric applications. For example, computing Gröbner bases of an algebraic system of small total degree (e.g. an ideal defined using three degree-two polynomials) can at times take many minutes on a current high-end workstation.

**Main Contribution:** We present efficient algorithms for exact representation, computation and geometric queries with algebraic numbers. Our contributions include:

- A hybrid approach to compute algebraic numbers combining techniques from symbolic computation based on exact arithmetic with floating point arithmetic. The resulting approach is reliable as well as efficient.

- A representation for points, curves and surfaces for efficient and reliable computations. In this paper, we restrict ourselves to algebraic sets in $\mathbb{R}^2$ and $\mathbb{R}^3$ only. The resulting approach can be extended to higher dimensions.

- We reduce computation of most geometric queries to computing signs of a series of determinants. The order of the matrices and size of matrix entries grows as a function of the degrees.

- The underlying algorithms have been implemented and used for efficient and reliable intersection computations between algebraic curves and surfaces. As compared to earlier implementations which produce reliable results, we are able to improve the overall performance by more than one order of magnitude.
Organization: The rest of this paper is organized in the following manner. We briefly survey related work in computational geometry, solid modeling and symbolic computing in Section 2. Section 3 discusses the main underlying problems and highlights various queries that come up in geometric algorithms dealing with algebraic numbers. We present efficient representations for points, curves and surfaces in Section 4 and highlight algorithms to compute with such representations in Section 5. An overall analysis of these algorithms is given in Section 6 and we discuss their implementation and performance in Section 7.

2 Related Work

There is considerable work in computational geometry, symbolic computing and solid modeling related to the problem of reliable computation.

2.1 Computational Geometry

The problem of robust and accurate computation has received considerable interest in the last few years. For general geometric algorithms, the emphasis has been on the use of exact arithmetic. Even for the geometric algorithms that can be written in terms of integer or rational arithmetic, the required bit-length of the integers typically exceeds the underlying machine precision. As a result, they require using software multi-precision arithmetic. However, a naive approach based on use of bignums can be rather expensive in practice and a number of authors have proposed techniques to speed up the paradigm of exact geometric computing. One approach, proposed by Fortune and Van Wyk [FV93], recommends the use of arithmetic filters. These filters safely evaluate a predicate in most cases, in order to avoid performing a more expensive exact computation. Fortune and Van Wyk used this approach in the LN package [FV93] and achieved considerable speed-up. These techniques have been appropriate for algorithms that use primitives of low algebraic total degree in two or three dimensions [FvW96]. Devillers and Preparata have investigated the theoretical behavior of some filters [DP97]. In degenerate or nearly degenerate cases, exact arithmetic has to be performed in full. LEDA [BKM+95] has support for exact arithmetic on algebraic numbers. However, it’s main limitation is that algebraic numbers can be created only from double
precision values and the nth root operation. There is no support to represent algebraic numbers as roots of an arbitrary polynomial. Yap [Yap97] has also advocated the use of exact arithmetic on algebraic numbers and shown that it is useful for a number of geometric algorithms. Many authors have proposed algorithms to reliably evaluate various predicates. In most cases, these predicates correspond to sign of an algebraic expression or a determinant. Designing a specialized implementation for evaluating such predicates can in many cases avoid the use of a general purpose multi-precision software.

In the last few years, a number of algorithms have been proposed to reliably evaluate the sign of a determinant [ABD+97, BEPP97, BY97, Cla92]. While these algorithms are general, their implementations have been restricted to determinants of small matrices of order (e.g. up to $6 \times 6$) or assume that the matrix entries are restricted based on the machine precision (e.g. 53 bits for IEEE double precision arithmetic). Schewchuck [She96] has described an adaptive and reliable implementation for low dimensional predicates with input entries specified using floating point numbers. Bajaj and Royappa have presented algorithms for parametric curves and surfaces based on finite precision representation [BR95].

## 2.2 Symbolic Computation

A number of algorithms have been proposed in the symbolic computation literature for computation and manipulation of algebraic numbers. These are based on resultants, Gröbner bases and root isolation techniques [Baj90, Buc89, Can88, CGT91, Col75, MC93]. Most of the earlier work has been on isolating roots of univariate polynomials [CK92]. More recently, Milne [Mil92] and Pedersen [Ped91] have extended Sturm sequences to multi-polynomial systems, which, along with resultants or Gröbner bases, can be used to isolate roots of multi-variate polynomial systems. In this paper, we present efficient algorithms based on multivariate Sturm sequences to represent algebraic numbers.

Most computer algebra systems have support for manipulating algebraic numbers. However, current implementations are extremely slow and computation of high degree algebraic numbers can take from a few minutes to a few hours on a current high end workstation. Techniques using bit-length estimates may, in the worst case, require bit-lengths which are exponential with respect to the degree of the algebraic functions [Can88, Yu92]. Techniques
based on *quantifier elimination* [Co75] can also be used to resolve some of the
queries arising in geometric algorithms. However, current implementations
[Hon92] are too slow for most geometric applications.

### 2.3 Geometric and Solid Modeling

The problem of CSG to boundary representation (“B-rep”) conversion has
been a fundamental problem in solid modeling [RV85, Hof89]. The primitives
of a CSG tree are typically composed of closed algebraic sets or solids
bounded by piecewise algebraic surfaces. However, the problem of robust
and accurate computation of the boundary is considered one of the difficult
problems in geometric and solid modeling [For96, KKM97]. It is important
that the computed B-rep be accurate, or at least topologically consistent, and
this can be jeopardized by even small amounts of error in the representation
of the model or in finite-precision computations. In this paper, we present
application of our algorithms to curve and surface intersections and show
how our representations can be used for accurate boundary computation.

### 3 Geometric Computations with Algebraic Numbers

A number is *algebraic* over a field $F$ if it is the solution to a polynomial
equation whose coefficients are in $F$. We will take $F$ to be the field $Q$ of
rational numbers. The set of algebraic numbers over $Q$ does not exhaust the
real numbers $\mathbb{R}$, but is sufficient for the geometric problems we consider.

A set of polynomial equations whose solution corresponds to a zero- or
one-dimensional algebraic set is given by:

\[
\begin{align*}
F_1(w_1, w_2, \ldots, w_n) &= 0 \\
F_2(w_1, w_2, \ldots, w_n) &= 0 \\
&\vdots \\
F_k(w_1, w_2, \ldots, w_n) &= 0.
\end{align*}
\]

where $k = n$ or $k = n - 1$. We will assume that their solution set contains no
excess components. In most applications, we are interested in evaluating all
the components of the algebraic set inside the region $D = [W_{(1,1)}, W_{(1,2)}] \times$
Formally, the functions $F_i$, $i = 1, 2, \ldots, k$, are the components of a vector function $F : D \to \mathbb{R}^k$, $D \subset \mathbb{R}^n$. The solutions to the problem are elements of $D$ that map to the zero vector under $F$. In this paper, our main emphasis is on algorithms where $n = 2$ or 3. However, the techniques presented are general and can be extended to higher dimensions. A number of queries in geometric algorithms can be formulated in this manner. Some of them include:

- **Intersection computations and representations of points, curves and surfaces (as algebraic sets):** This arises in the computation of Voronoi regions of lines, planes and polyhedra, boundary computations on curved solids, and arrangements of non-linear primitives. In each of these applications, the zero-dimensional algebraic set corresponds to the intersection points of three algebraic surfaces. For example, while computing Voronoi regions of polyhedra, we compute the zero-dimensional intersection of three quadric surfaces. In boundary computation, given two sets of piecewise algebraic surfaces representing closed solids, features of the new solid contain vertices which are obtained by computing the intersection of three surfaces.

- **Orientation of a point, inside/outside tests with respect to an algebraic set or a region of space bounded by piecewise algebraic sets:** Orientation tests are used in the algorithms mentioned above as well. In the case of Voronoi region computation, a frequent operation is to test whether a vertex lies inside or outside a sphere. Component classification in boundary evaluation algorithms involves checking if a point lies inside or outside a solid whose boundary is composed of piecewise algebraic surfaces. Sign queries are also used to identify degenerate geometric situations.

Consider the class of algorithms that perform these computations in double precision floating point arithmetic. The results of this algorithm will not always match the output specification given above. Therefore, the output is usually modified as follows. Along with the set of equations, the algorithm requires the definition of a vector norm $\| \cdot \|$ on $\mathbb{R}^n$ and a small positive constant $\delta$ (at least machine precision) as additional inputs. Then the output set consists of all those elements $x \in D$ such that $\| F(x) \| < \delta$. This kind of specification has two problems. Firstly, while the backward error in these
methods is bounded, it does not say anything about the forward error. The computed solution could be far away from the actual solution. Secondly, it is possible that some solutions are missed because of their proximity to other solutions. Furthermore, the inaccuracies due to round-off errors in finite precision may easily lead to wrong sign evaluation of the algebraic predicates while performing orientation tests.

We use exact arithmetic to circumvent such problems. However, exact arithmetic on high degree algebraic numbers can be slow. In this paper, we propose a hybrid approach that takes advantage of the efficiency of floating point arithmetic while maintaining the accuracy and consistency of the solutions using exact rational arithmetic. We represent solutions (algebraic numbers) of zero-dimensional algebraic systems (vertices or points in geometric applications) using rational boxes. These boxes isolate the algebraic points and can be resolved (the box tightened) in a lazy manner. Isolation and resolution are accomplished using multivariate Sturm sequences [Mil92]. In practice, root isolation can be expensive and its cost is directly proportional to the number of Sturm sequence evaluations. To minimize evaluations, we evaluate the points using double precision arithmetic first. The results are verified using Sturm sequences in rational arithmetic. We evaluate these intervals to a precision that is enough to isolate the roots, and as the computation demands, they are refined later. In most cases, results of floating point arithmetic are enough to guarantee the correct solution. We detect the few cases that pose a problem with finite precision and identify methods to recover the correct answer. This approach has the following advantages:

- Floating point arithmetic is used whenever possible.
- The cost of performing the rational arithmetic checker is directly related to the separation of roots or degeneracy in the computation. For example, more Sturm sequence calls are needed to separate two points that are very close than when they are well apart.
- The results obtained are geometrically accurate and topologically consistent with the exact solution.

Hybrid approaches using a combination of exact arithmetic and floating point arithmetic have been used earlier in computational geometry. However,
their use has been restricted to inputs dealing with linear primitives or low-degree non-linear primitives [FvW96]. For example, the orientation of the intersection point of 3 planes with respect to a fourth plane reduces to computing the sign of a $4 \times 4$ determinant [FV93]. However, in the non-linear case, the orientation test is equivalent to a quantifier elimination problem over algebraic sets. Our representation of zero-dimensional algebraic sets allows us to evaluate the orientation test effectively by testing for intersection between rational boxes and an algebraic set.

4 Representation of Geometric Primitives

In this section, we describe representations of points, curves and surfaces involving algebraic numbers. Based on these representations, we present efficient algorithms for different queries.

4.1 Representation of Points

Multivariate St"urm sequences allow the representation of a point $p \in \mathbb{R}^n$, whose coordinates are algebraic numbers, as follows. An algorithm specifies $p$ implicitly, as one solution to a set of $n$ polynomial equations. Using multivariate St"urm sequences, the solutions to the system (within some relevant region) are isolated: a set of boxes is constructed in $\mathbb{R}^n$ such that each solution lies in exactly one box, and each box contains exactly one solution. Such a box is considered as a simple approximation to the point. The boxes are axis-aligned and have rational coordinates.

When evaluating a geometric query involving the point $p \in \mathbb{R}^n$, the algorithm attempts to answer the query based on the rational box known to contain $p$. If the results are ambiguous, the approximation to $p$ is resolved by subdividing the box (into $2^n$ equal-size sub-boxes), and repeating until the query can be answered unambiguously. In this manner, we start with a low-precision approximation to $p$ and improve the approximation only as needed. Thus, we have a form of lazy evaluation of geometric predicates.

Some queries cannot be posed on boxes alone. For example, an algorithm may have to determine conclusively whether two points $p$ and $q$ in $\mathbb{R}^n$ are equal. In this case, we can compute the intersection of the intervals containing $p$ and $q$ and formulate a new system of equations (obtained by taking
sum of the squares of the original algebraic equations used to define \( p \) and \( q \). The algorithm uses Stürm sequences to check whether the new system has any solution in the intersection of the solutions.

### 4.2 Representation of Curves and Surfaces

We represent an algebraic (implicit) curve in \( \mathbb{R}^2 \) with a polynomial \( f(s,t) \) and two endpoints. Each endpoint \( p \) is represented implicitly as the intersection between \( f(s,t) \) and some other curve \( g(s,t) \), along with a rational two-dimensional box containing \( p \) and no other intersections of \( f(s,t) \) and \( g(s,t) \). The second curve \( g(s,t) \) is either a line representing the edge of a region of interest, or some curve which arises during the algorithm.

Rational parametric surfaces in our algorithms are represented with parametric functions and a corresponding domain. We make use of an implicit form of these surfaces when intersecting them. The implicit form can be computed efficiently using implicitization algorithms [MC92]. The rational surfaces may be trimmed by algebraic curves in the domain. A rational surface is then represented by a rational function \( X: \mathbb{R}^2 \to \mathbb{R}^3 \), together with a collection of trimmed regions in \( \mathbb{R}^2 \). Each trimmed region is represented by a circular sequence of algebraic curves and curves intersections (two-dimensional algebraic points). An “untrimmed” patch is represented with a single trimmed region (say a unit square or a rectangle) with four linear boundary curves. Other trimming curves arise from surface intersection; if our patch has parametric form \( X(s,t) \), and an intersecting patch has implicit form \( F(x,y,z) = 0 \), then trimming our patch to the intersection curve induces the trimming curve \( f(s,t) = F(X(s,t)) \).

Each trimming curve, and each vertex between a pair of trimming curves, has an image in \( \mathbb{R}^3 \). Many queries about a vertex can be posed and answered in the domain of the surface. But some queries depend on a vertex’s position in \( \mathbb{R}^3 \). For such queries, trivariate Stürm sequences are used to isolate the point inside a three-dimensional rational box. The two-dimensional representation of \( p \) and the three-dimensional representation of \( X(p) \) are not completely independent,\(^1\) but two-dimensional Stürm sequences alone cannot guarantee the isolation of \( X(p) \).

\(^1\)For example, the two-dimensional box surrounding \( p \) is the domain of a small patch under \( X(s,t) \); a three-dimensional box containing the Bézier control points (obtained by representation of the parametric function in Bernstein basis) of this patch contains \( X(p) \).
These representations are sufficient for many applications including the CSG-to-boundary-representation problem and computation of Voronoi regions. For the first, we construct closed 2-manifolds; for the second, a non-manifold surface. Yet both can be constructed and represented from trimmed rational surfaces, where a surface’s trimming curves always occur at intersections with other surfaces, and the curves meet at vertices where three (or more) surfaces intersect.

Our approach is distinguished from other exact-arithmetic approaches in several ways. First, we approximate only algebraic numbers with rectangular boxes; rational numbers are represented exactly. A consequence is that the overall algorithm must be aware of the representation of the numbers. This makes algorithms more complicated; but we believe this is an appropriate price to pay, since the ultimate behavior of the algorithm depends on the quality of approximation. Another distinguishing characteristic is that we use floating-point arithmetic to speed up certain calculations, but floating-point estimates are always eventually verified with rational estimates.

5 Efficient Geometric Computations

In the last section, we have described our representations for points. They are based upon finding either the intersection point between two algebraic plane curves within some interval, or the intersection point of three surfaces within some three-dimensional box. This fundamental operation, corresponding to root isolation and refinement, accounts for the vast majority of our time in an application based upon these representations. In this section we discuss efficient algorithms for computing with these representations.

5.1 2D Computations

Most of the 2D operations involve solving for zeros of two bivariate polynomials. These operations arise frequently in boundary evaluation and Voronoi region computation algorithms. Multivariate Sturmi sequences, as described in [Mil92], are used to determine the zeros.

The basic method for locating the zeros of two bivariate polynomials, \( f(s,t) = 0 \) and \( g(s,t) = 0 \), is to consider a third polynomial, \( h(s,t,u,a,b) = u + (s-a)(t-b) = 0 \), and eliminate \( s \) and \( t \) from the combined system of three
equations. This leaves one with a single polynomial, \( R(u, a, b) = 0 \). Treating \( a \) and \( b \) as constants, the algorithm generates a univariate Stürm sequence for the polynomial \( R(u) = 0 \). Counting the number of sign permanencies in this sequence at \( u = 0 \), one has the number of real roots, \((a, b)\) such that \((s - a)(t - b) > 0\), plus some constant count related to the number of complex roots. Let \( N(a, b) \) be the number of sign permanencies in the sequence associated with the point \((a, b)\). To find the number of roots in an interval \( s \times t = [\alpha_s, \beta_s] \times [\alpha_t, \beta_t] \), we compute \( \frac{1}{2}(N(\alpha_s, \beta_t) - N(\alpha_s, \beta_t) - N(\beta_s, \alpha_t) + N(\beta_s, \beta_t)) \). Let us consider each of these steps in more detail.

The first step in this computation is to eliminate \( s \) and \( t \) from the three polynomials \( f = 0 \), \( g = 0 \), and \( h = 0 \). Let the maximum degree of \( s \) in \( f(s, t) \) be \( m_1 \), and the maximum degree of \( t \) in \( f(s, t) \) be \( n_1 \). Similarly define \( m_2 \) and \( n_2 \) for \( g(s, t) \). We use the following theorem, as given in [Mil92] for resultant computation.

**Theorem:** The resultant of \( f(s, t) \), \( g(s, t) \) and \( h(s, t) \) is given by

\[
\text{Res}((\text{Res}(f(s, t), h(s, t), s)), (\text{Res}(g(s, t), h(s, t), s)), t)_{u^{m_1+m_2}}
\]

These three resultants can be computed using the Sylvester formulation; in particular, the inner two resultants are computed by simple elimination and substitution.

To efficiently compute the resultant we substitute the numeric values for \( a \) and \( b \) in the third equation. Although this means that we must perform this computation multiple times (once for each substituted value for \((a, b)\)), it allows us to avoid computing the large symbolic polynomial, \( R(u, a, b) \).

The Sylvester resultant can be set up as the determinant of a matrix where each term is a polynomial in terms of \( u \). However, this would involve symbolic manipulations, which can be very expensive to perform. Instead, we set up the problem as an interpolation problem [MC93]. We use a Vandermonde matrix formulation, where we solve numerical Sylvester resultants to determine the right-hand side of the system. Since we can compute the maximum degree of \( u \) in \( R(u) \) using Newton’s polytope (mixed volumes) method, we know the size of the Vandermonde system. Thus, the elimination of \( t \) becomes a purely numerical computation.

At this point we are left with a single polynomial in \( R(u) \), called the volume function, for which we need to determine the Stürm sequence. There are two main ways we can approach this: Euclid’s method or subresultant
sequences. Using Euclid’s method is the straightforward, “standard” way of computing Sturm sequences. The Euclid method approach involves computing the gcd of \( R(u) \) and \( R'(u) \). The intermediate terms form the Sturm sequence. Notice that in general, the length of the Sturm sequence is equal to the degree of \( u \) in \( R(u) \). This method can be quite efficient, particularly for lower degree polynomials.

For higher degree polynomials, however, Euclid’s method can become quite inefficient. This is due to the fact that the size of the coefficients needed in each successive polynomial grows exponentially [Knu81]. This is a fairly well-studied problem in polynomial gcds, and has led to the development of the subresultant polynomial remainder sequence algorithm [BT71]). With this approach, the coefficients of the terms of the polynomial sequence can be formulated as the determinant of a submatrix of a Sylvester matrix. Notice that since we evaluate the polynomials at \( u = 0 \), we are only interested in the constant terms of each polynomial in the sequence. Furthermore, since we only count the sign changes between successive polynomials, we are interested only in the signs of determinants of a set of matrices.

In summary, the computation of the zeros of two bivariate polynomials plane curves boils down to the construction and solution of a Vandermonde system, followed by the evaluation of the signs of several determinants. When the curves being intersected are of low degree, the time taken in constructing and solving the Vandermonde system dominates the time, however, the time for evaluating the signs of determinants rapidly dominates for higher degree curves.

### 5.2 3D Computations

In the 2D case, we constructed a system of three equations in three variables—the two given equations in \( s \) and \( t \), with a third equation \( u + (s - a)(t - b) \), where \( u \) is the third variable and \( a \) and \( b \) are numerical constants. For the intersection of three surfaces in \( \mathbb{R}^3 \), we construct a system of four equations in four variables—the three given equations in \( x \), \( y \) and \( z \), with a fourth equation \( u + (x - a)(y - b)(z - c) \). In the former case, we were able to eliminate \( s \) immediately, since \( s \) is a rational function in \( t \) and \( u \). In the latter case, we do not have this luxury and must resort immediately to resultants. As with the 2D case, we must eliminate \( x \), \( y \), and \( z \) from this system of four polynomials. One option is to eliminate variables one-by-one with successive
Sylvester resultants. The disadvantage here is that this process will amass a large extraneous factor.

A more efficient approach is to use the Macaulay resultant by eliminating \( x, y \) and \( z \) in one step, with no extraneous factors [Mac02]. The Macaulay formulation expresses the resultant as a ratio of two determinants. The immediate drawback here is that the order of the matrices grows with the degrees of the input equations. For example, given three quadrics, the Macaulay numerator is the determinant of an \( 84 \times 84 \) matrix, with entries that are polynomials in \( u \). However, most of the entries are constants; only a \( 8 \times 8 \) minor of that matrix has entries containing \( u \), and these are linear in \( u \). By taking advantage of this sparsity, we will be able to efficiently compute the resultant.

Once the resultant volume function has been computed, we proceed as before. We substitute eight different numerical values for \((a, b, c)\), at the vertices of a cube, and the Sturm sequence returns the number of roots of the system inside the cube.

### 5.3 Signs of Determinants

The efficiency of our algorithms is primarily limited by the speed with which we can compute the sign of the determinant of an integer matrix. Typically the entries of these matrices are bigints whose magnitude grows quadratically with the degrees of the input equations. Computing the determinant with floating-point arithmetic may produce incorrect results. A floating-point filter, as in LEDA [BKM+95] returns one of \(+1, -1\), or \textsc{No IDEA} for the sign of the determinant. We have found that this significantly speeds up computation for determinants of small matrices, but usually returns \textsc{No IDEA} for the larger determinants that arise in higher-degree Sturm computations. Straightforward determinant computation with bigints is impractical, as the bit-length of the intermediate quantities grows exponentially with the size \( n \) of the matrices, while the determinant's bit-length is \( O(n(\log n + b)) \) (\( b \) is the bit size of the original matrix entries) according to Hadamard's bound.

A better solution is to compute the determinant several times over various finite fields, and reconstruct the actual determinant from these residues. The finite fields are chosen such that hardware arithmetic can be used directly—computation is done modulo primes on the order of \( 2^{16} \) (using 32-bit integer arithmetic) or the order of \( 2^{27} \) (using double-precision floating-point).
One has a choice of reconstruction methods, but the overall running time is generally proportional to the number of fields over which the determinant is computed. To reconstruct the determinant completely (and its sign exactly), it is necessary to compute residues over sufficiently many fields so that the total number of bits in all the residues is as large as the bit-length of the actual determinant. Hadamard’s bound is the best known bound on the bit-length of the determinant, in terms of the bit-lengths of the entries of the matrix. A reasonable strategy is to compute Hadamard’s bound for the given matrix, and use this bound to decide on the number of fields over which the determinant will be taken. This appears to be the approach used in LiDIA [BBP95].

An alternative to full reconstruction of the determinant is Newton’s iterative reconstruction [MC93] which terminates early for determinants which are much smaller in magnitude than Hadamard’s bound predicts. With extremely low probability, Newton’s algorithm mistakenly terminates and produces a wrong answer. For a large matrix with a small determinant, Newton’s algorithm lets us compute the determinant over as few as two finite fields, instead of several hundred.

Recently, a new reconstruction method has been proposed by Brönnimann et al. [BEPP97], where the reconstruction process gives an early exit when the determinant is fairly small. It is effective for matrices with short entries, but it seems to require that the determinant be computed over all of the fields prescribed by Hadamard’s bound before reconstruction begins, and therefore does not show much improvement over full reconstruction for matrices with large entries. We have implemented these different algorithms for computing sign of determinants and results are presented in Section 7.

5.4 Incorporation of Floating-Point Arithmetic

As described, the computations of roots can be very slow. Although more efficient implementations for underlying routines, such as determinant sign evaluation, can speed up the computation process, the size of the numbers required and the additional operations needed for exact computation will cause the overall computation to be slow. An inexact approach, such as one using floating-point methods, will be much faster. We therefore want to combine the accuracy of the exact approach with the speed of floating-point hardware available on current machines.
One way that this can be done is to compute a floating-point approximation for each computation, and then use an exact approach to verify that the floating point method was correct. For example, to find roots of a univariate polynomial, one could use a standard floating-point based method (e.g., Newton’s method) to get an approximate solution, \( \alpha \). Then, one can choose an interval, \([\alpha - \epsilon, \alpha + \epsilon]\), and use an exact method to test whether or not the true root is within that interval.\(^2\) If so, then the root has been rapidly determined to a high precision with far fewer exact calls than would be necessary in a bisection approach. If the root does not lie in the interval, then one can expand the interval and use an exact test until the root does lie inside the interval. Then, the standard bisection approach can be used to determine the root to the desired precision. As a root is determined to higher and higher precision, at some point the floating-point value may fall outside the interval. At this point, it is necessary to switch to performing only the exact computation.

In most cases, the floating-point method gives us accurate results to several digits, so a number of steps are saved. Even if the floating-point approximation is totally off, the problem never becomes worse than the original “find a root in this domain” problem, and the extra time spent will be minor compared to the cost of computing the root by bisection. For extremely ill-conditioned cases, the floating point approximation may be completely wrong or only provide a couple of digits of accuracy. In most cases, however, this approach will save a considerable amount of time by reducing the number of exact computations necessary.

The same idea is used in isolating roots. In this case, we determine approximations to the original roots as well as distance to the nearest root. Thus, we also have an approximate value for the width of interval needed to isolate each of the roots. Again, in the worst case, we are back where we started—isolating the roots by repeatedly bisecting the domain.

Although this discussion has been about roots in one dimension, the same principle holds for intersections in higher dimensions. For example, in two dimensions, the intersection point between two curves can be determined in floating-point by posing it as a generalized eigenvalue problem (see [MC93]). The approximations of the intersection points thus obtained can then be used\(^2\) In practice, rather than computing \( \epsilon \), we simply choose it so as to minimize damage to the bit-lengths of \( \alpha \pm \epsilon \).
to rapidly isolate the exact roots and determine them to whatever precision necessary.

In effect, this approach means that we are performing floating point arithmetic, but using exact methods to verify our answers. Only when floating-point methods break down do we rely entirely on exact computation.

6 Analysis

In this section, we present an analysis of the number of steps we take to isolate a root, obtained by the intersection of two planar curves \( f(s, t) \) and \( g(s, t) \), to within a precision of \( \epsilon = 2^{-\delta} \). We also assume that \( f(s, t) \) (\( g(s, t) \)) is of degree \( m_1 \) (\( m_2 \)) in \( s \) and \( n_1 \) (\( n_2 \)) in \( t \). Let

\[
f(s, t) = \sum_{i=0}^{m_1} \sum_{j=0}^{n_1} a_{ij} s^i t^j \tag{1}
g(s, t) = \sum_{i=0}^{m_2} \sum_{j=0}^{n_2} b_{ij} s^i t^j \tag{2}
\]

and let \( D \) be the maximum number of bits to represent the coefficients (i.e., \( \max(|a_{ij}|, |b_{ij}|) < 2^D \)). During the computation of the two-dimensional Stürm sequence, we adjoin another polynomial of the form \( h(u, s, t) = u + (s - a)(t - b) \). Here \((a, b)\) represents a corner of the rectangular box inside which we are finding the roots. We can, thus, bound the bit lengths of \( a \) and \( b \) by \( \delta \).

In the first step, we eliminate \( s \) from our equations, as described in section 5.1. This results in two polynomials \( f_h(t, u) \) and \( g_h(t, u) \) with leading terms \( t^{m_1} u^{n_1} \) and \( t^{m_2} u^{n_2} \) respectively. The bit size of the coefficients of these polynomials are bounded above by \( \Delta = D + \delta \max(m_1, m_2) \).

In order to obtain the volume function (in \( u \)), we eliminate \( t \) from \( f_h \) and \( g_h \) by setting up the Sylvester matrix. Instead of symbolically expanding the Sylvester determinant, we solve a \((N + 1) \times (N + 1)\) linear Vandermonde system, where \( N \) is the degree of the volume function. In order to set up the right-hand side of this system, we have to solve \((N + 1)\) numeric Sylvester determinants of order \( n = m_1 + n_1 + m_2 + n_2 \). The total cost of this step (given that Bareiss's version of Gaussian elimination [Win96] takes about \( n^3/2 \) operations) is

\[
\frac{(N + 1)}{2} [(N + 1)^2 + (m_1 + n_1 + m_2 + n_2)^3].
\]
The asymptotic cost clearly depends on the value of \( N \). A naive estimate given \( f_h \) and \( g_h \) from the Sylvester resultant (after factoring out extraneous factors) would indicate that \( N = m_1 n_2 + m_2 n_1 + m_1 m_2 \). However, since \( h(u, s, t) \) is linear in \( u \), the maximum degree of the volume function is given by the total number of common roots of \( f(s, t) \) and \( g(s, t) \). The total number of common roots in this case is \( m_1 n_2 + m_2 n_1 \). Given this estimate on \( N \), the total cost is clearly dominated by \((N + 1)^3/2\). If \( m_1 = n_1 = m_2 = n_2 = k \), it gives a \( O(k^6) \) cost.

Now we estimate the size of the coefficients in the volume function. It is quite obvious that each row operation in Gaussian elimination roughly doubles the bit complexity. Therefore, at the end of the entire algorithm the bit lengths of the coefficients of the volume function could be as high as \( O(2^{N+1} \Delta) \). Factoring out the gcds for these coefficients could potentially be very expensive. However, Bareiss’s modification of this method [Win96] identifies divisors of rows in the elimination process without actually having to compute gcds. In this method, instead of doubling bit size during each row operation, the increase is by \( \Delta \) each time. Thus the bit size of the coefficients of the volume function is roughly \( O((N + 1)\Delta) \).

The final step in the root isolation procedure is to compute the Sturm sequence of the volume function and count the number of sign changes in the constant term of each polynomial in the sequence. We use subresultants to compute the sign of the constant term in the polynomial remainder sequence. The main advantage of using this method over Euclid’s polynomial gcd method is that the latter has to compute all the terms of every polynomial and has to deal with significant growth in the coefficients. Computing the sign of the constant term involves computing the sign of a matrix determinant. The entries in the matrix determinant are of size \( O((N + 1)\Delta) \) bits.

We evaluate \( N \) matrix determinant signs of size 3, 5, 7, \ldots, 2N - 1. Currently, the theoretically-best method to evaluate determinant signs in the literature is by Brönnimann, Emiris et. al [BEPP97]. Their method of recursive relaxation of moduli enables them to carry out sign determination by using only floating point computations in single precision. Their method finds the sign of a \( n \times n \) determinant in \( O(n^4 \log n) \) single precision operations. The total cost to evaluate the roots from the Sturm sequence is \( O(N^5 \log N) \).

However, their algorithm has a restriction that the matrix entries should be integers exactly representable in 53 bits. In our applications, we encounter integers 300 bits and larger. Our implementation of an extension of their
algorithm to arbitrary sized integers is slower than computing the exact determinant in bignum arithmetic using a library like LiDIA (see section 7.4). For this analysis, however, we use an $O(N^{5.1} \log N)$ bound on the number of operations.

It is clear that the dominating cost in the root isolation algorithm is that of determinant sign evaluation. Hopefully, with better algorithms for this problem, we can resolve non-linear geometric queries much more efficiently.

7 Implementation and Performance

In this section we discuss the performance of our implementation of the representations and their associated computational routines. We have implemented routines for finding exact roots in 1-D and 2-D, as well as portions of a boundary computation program.

Our implementation involves representations for surfaces, curves, and points, and is based upon a set of polynomial libraries. We use the LiDIA library to handle bigint and bigrational computation. Except where stated otherwise, all 2D computations use the LiDIA bigint matrix class to compute the signs of determinants for Stürm calculations (as opposed to using Euclid’s method or a different sign-evaluator). Also, unless stated otherwise, the hybrid floating-point/exact computation approach is used (as opposed to a strictly exact approach). All times are in user-seconds on an SGI Onyx workstation with a 200MHz R4400 processor.

A Stürm computation breaks into two steps:

- The generation of the volume function $R(u)$: constructing Sylvester resultants and solving the Vandermonde system

- The determination of the number of sign permanencies: Euclid’s gcd algorithm (low-degree systems) or subresultant formulation in terms of signs of determinants (higher-degree systems).

Table 2 gives the breakdown of the time spent in both parts of the computation for five examples. The examples are for pairs of curves of varying degrees and varying coefficient bit lengths.

The bulk of the time in intersecting two surfaces is spent in these two steps. For low-degree systems, about 90% of the time is spent in the Vandermonde solver, and the remainder largely in Euclid gcds. For higher-degree
systems, subresultants are faster than computing gcds, but the subresultant time begins to dominate the computation, and the program spends more than 97% of its time computing signs of determinants in the subresultant step.

### 7.1 Univariate Root Finding

Table 1 presents some timing results for univariate root finding. The examples are from the family of *Wilkinson polynomials*

\[ W_n(x) = \prod_{i=1}^{n} (x - i). \]
Notice that for this notoriously ill-conditioned polynomial, the exact methods determine points to an accuracy unattainable with strictly floating-point methods. For the degree 20 case, for example, double-precision floating-point arithmetic can only provide two digits of accuracy. This example shows the usefulness and reliability of exact arithmetic when dealing with polynomial roots.

### 7.2 Resultant Computation

The first step of our routine for plane curve intersections is to determine the volume function, $R(u)$. This involves computing a Sylvester resultant by solving a Vandermonde system. Timings for construction and solving the Vandermonde system are given in Table 2. Using a Gröbner Basis implementation to find $R(u)$ was found to take more than 100 times as long in some cases.

<table>
<thead>
<tr>
<th>System</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
<th>Example 4</th>
<th>Example 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree of curves</td>
<td>$(2, 1, 2)$</td>
<td>$(4, 2, 6)$</td>
<td>$(4, 4, 4)$</td>
<td>$(3, 4, 4)$</td>
<td>$(3, 1, 3)$</td>
</tr>
<tr>
<td>Intersections</td>
<td>$(1, 2, 2)$</td>
<td>$(3, 2, 5)$</td>
<td>$(4, 4, 4)$</td>
<td>$(4, 4, 4)$</td>
<td>$(1, 3, 3)$</td>
</tr>
<tr>
<td>Volume function</td>
<td>Degree of $R(u)$</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Bits in curve coefficients</td>
<td>83</td>
<td>189</td>
<td>312</td>
<td>482</td>
<td>208</td>
</tr>
<tr>
<td>Times (sec)</td>
<td>Floating-point approximation</td>
<td>0.07</td>
<td>0.1</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Counting permanencies (gcd or subres.)</td>
<td>0.77</td>
<td>0.5</td>
<td>4.0</td>
<td>55.2</td>
<td>4.9</td>
</tr>
<tr>
<td>Total</td>
<td>0.90</td>
<td>8.0</td>
<td>1600.0</td>
<td>2036.0</td>
<td>7.8</td>
</tr>
<tr>
<td>Speedup due to FP approx.</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: Timing breakdown of five example systems in $\mathbb{R}^2$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
<th>Example 4</th>
<th>Example 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euclid’s gcd algorithm</td>
<td>0.06</td>
<td>29</td>
<td>2140.0</td>
<td>4140.0</td>
<td>7.8</td>
</tr>
<tr>
<td>Subresultants (LiDIA determinants)</td>
<td>1.12</td>
<td>15.8</td>
<td>1550.7</td>
<td>1980.7</td>
<td>55.9</td>
</tr>
</tbody>
</table>

Table 3: Counting permanencies: comparison of two methods.
7.3 Subresultants vs. Euclid’s Method

As mentioned earlier, subresultant sequences are useful mainly in controlling the growth of coefficients in later terms in the Sturm sequence. When the Sturm sequence is not too long, and the starting coefficients are not too big, however, Euclid’s method can be faster. Table 3 presents the relative time taken by each of the two methods on our example cases.

7.4 Signs of Determinants

Table 4 shows running times of three different sign-of-determinant routines. The matrices are drawn from the sub-resultants in the computation of the intersection points of two algebraic plane curves, each of which is of total degree 4 in $s$ and $t$. For each value of $n$, an $n \times n$ matrix was tested, whose largest entry has a bit-length between 300 and 350 bits. The LiDIA time represents a call to `bigint_matrix::det()`. The Relax time is our modification to Sylvain Pion’s relaxed-multiplication code [BEPP97]. (The modification is to allow arbitrary integers, rather than just 53-bit integers, in the matrix.) The Newton time is the same code as Relax, but using Newton’s iterative, probabilistic reconstruction.

To the best of our knowledge, LiDIA’s speed comes from the use of assembly-language code and other such optimizations.

7.5 Floating-Point Method Incorporation

In section 5.4 we discussed methods for incorporating floating-point methods to speed up the computation. Table 2 shows that speedups of five to eight are to be expected, when the roots are resolved to boxes of size $1/1000$ (as they are in our examples). The floating-point methods show the greatest improvement when the roots are resolved to the same accuracy which the floating-point computation exhibits. In these cases, we have observed speedups of one to two orders of magnitude.

7.6 Curves and Surfaces

We have tested our implementation on a few patch-patch intersections which would arise in a boundary computation problem. These intersections involve
intersecting two surfaces to find the intersection curve in the domain of each, finding the points at which that intersection curve hits the patch boundary (edge points) - a 1-D intersection problem, and finding the turning points of the intersection curve (in s and t) within the patch domain - a 2-D intersection problem. The time taken for these operations in a few patch-patch intersections is given in table 5.

### 7.7 Comparison with other approaches

Several other approaches to exact geometric computation have been attempted. The C++ library LEDA has a class `Real`, but it does not currently support all algebraic numbers—only those generated from the rationals by arithmetic and $n$th-roots. The LiDIA library has a class `alg_number`, which represents an algebraic number as a rational linear combination of primitives. For example, one can perform arithmetic over the field $\mathbb{Q}(\sqrt{2})$. Neither LEDA’s representation nor LiDIA’s supports geometric queries on roots of
All of the algorithms we present here can be implemented on general-purpose computer algebra systems, which provide support for algebraic numbers. They are normally based on Gröbner bases and root isolation. In Stürm computation, the volume function $R(u)$ can be found by taking a Gröbner basis of the polynomial system. However, even efficient implementations of these algorithms perform poorly on moderate-sized systems, because of their asymptotic complexity. The volume function in Example 2 in table 2 took eight minutes to compute using Mathematica's `GroebnerBasis` function—two orders of magnitude slower as compared to our resultant computation algorithms.

Most implementations of root isolation have been restricted to univariate polynomials and we are not aware of any extensions to multi-variate systems. The algebraic toolkit, developed by Rege and Canny [Reg95], has support for univariate root isolation algorithms. Most quantifier elimination implementations, (e.g. SAC-2 or the one presented in [Hon92]) are quite slow as compared to our implementation.

There is considerable work on numeric approximation to algebraic numbers based on homotopy methods [MSW89] or combination of resultant and eigenvalue methods [Man94]. However, they use finite precision arithmetic and cannot guarantee accurate or reliable results.

Overall, our algorithms and implementations for computing algebraic numbers and using them for geometric queries are more than one order of magnitude faster as compared to earlier implementations that produce reliable results.

<table>
<thead>
<tr>
<th>First Surface Type</th>
<th>Second Surface Type</th>
<th>Number of Edge Points</th>
<th>Number of Turning Points</th>
<th>Time to Compute (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylinder</td>
<td>Cylinder</td>
<td>4</td>
<td>4</td>
<td>49.7</td>
</tr>
<tr>
<td>Sphere</td>
<td>Plane</td>
<td>4</td>
<td>1</td>
<td>4.6</td>
</tr>
<tr>
<td>Torus</td>
<td>Plane</td>
<td>6</td>
<td>3</td>
<td>120.6</td>
</tr>
</tbody>
</table>

Table 5: Examples from patch-patch intersections.
8 Conclusion and Future Work

In this paper we have presented algorithms to represent algebraic numbers and used them to resolve geometric queries accurately and efficiently. Based on that, we have discussed how we can represent points, curves, and surfaces which involve algebraic numbers in a consistent manner within a geometric application. Also, we have described techniques that are used in computing with these representations and allow us to compute answers more efficiently. In particular, we combine techniques from symbolic computation based on exact arithmetic with floating point arithmetic for fast and reliable computations. We also highlight results from our implementation and its applications to curve and surface intersections. As compared to earlier implementations that produced reliable results, our algorithm is more than one order of magnitude faster.

As we deal with high degree algebraic numbers (say solutions of algebraic equations, whose total degree or Bézout bound is more than 7 or 8), a very significant fraction of the overall time is spent in computing signs of a series of determinants. Although our algorithm uses lazy techniques to isolate the roots to minimal precision, the resulting coefficients can still be very large. We evaluated different algorithms and public domain implementations for computing signs of determinant and empirically observed that LiDIA’s implementation is the fastest. However, more work is needed to improve the performance of sign determination algorithms for such matrices. We also plan to look at other applications, besides intersection computations and boundary evaluations, of these algorithms. These include computation of generalized Voronoi diagrams of lines and planes as well as medial axis transform of a polyhedron.

References


