MAPC: A library for Efficient and Exact Manipulation of Algebraic Points and Curves*

John Keyser Tim Culver Dinesh Manocha Shankar Krishnan†
Department of Computer Science
University of North Carolina, Chapel Hill, NC 27599-3175
{keyser,culver,dm}@cs.unc.edu, krishnas@research.att.com
http://www.cs.unc.edu/~geom/MAPC/

Abstract:
We present MAPC, a library for exact representation of geometric objects—specifically points and algebraic curves in the plane. Our library makes use of several new algorithms, which we present here, including methods for finding the sign of a determinant, finding intersections between two curves, and breaking a curve into monotonic segments. These algorithms are used to speed up the underlying computations. The library provides C++ classes that can be used to easily instantiate, manipulate, and perform queries on points and curves in the plane. The point classes can be used to represent points known in a variety of ways (e.g. as exact rational coordinates or algebraic numbers) in a unified manner. The curve class can be used to represent a portion of an algebraic curve. We have used MAPC for applications dealing with algebraic points and curves, including sorting points along a curve, computing arrangement of curves, medial axis computations, and boundary evaluation on curved primitives. As compared to earlier algorithms and implementations utilizing exact arithmetic, our library is able to achieve more than an order of magnitude improvement in performance.

1 Introduction

A common assumption in the design of geometric algorithms is that all points can be easily defined and manipulated. Such an assumption is usually a result of relying on the "real RAM" model of computation [For95], in which all arithmetic operations are exact and take constant time. No computer implements this model. Implementors must sacrifice either exactness (as in floating-point) or constant time performance (as in multiprecision arithmetic).

For those interested in achieving accuracy and robustness in an implementation, the choice is usually to use exact arithmetic while sacrificing speed. For geometric programs which rely on only linear geometric primitives (i.e. line segments and their intersections), exact rational arithmetic is enough to handle all necessary numbers. When one considers points and curves defined by rational polynomial functions, however, it becomes necessary to handle real algebraic numbers, since intersections of such curves are points whose coordinates are real algebraic numbers. However, representing and manipulating real algebraic numbers requires more complicated data structures and algorithms than rational numbers.

A further drawback in dealing with points whose coordinates are algebraic numbers comes in dealing with them in a unified manner. For example, a representation that is appropriate for storing points with algebraic coordinates is often an overkill when a coordinate is known to be a rational number. In any practical system, it is desirable to have a unified approach where all points can be represented in the simplest known form.

Other difficulties arise in representing curves. While line segments can be represented using only two endpoints, far more information is necessary to represent a curve segment. Algebraic curves may have multiple components and cannot necessarily be expressed as a rational parametric curve. In many applications, one is not interested in the entire curve over the entire space, but rather some trimmed portion of the curve within a particular region of space. Dealing with points and curves can be a significant bottleneck in efficient and robust implementations of geometric algorithms.

The need for effective manipulation of algebraic points and curves comes up in several applications. These include boundary evaluation algorithms on NURBS and algebraic primitives [KKM97, Hof89], computing the medial axis or internal Voronoi regions of a polyhedron [SP95, CKM98], and robot motion planning [Can87]. Most of the earlier algorithms and libraries do not provide adequate support for manipulating algebraic points and curves.

In this paper, we present efficient algorithms and a library, MAPC, for efficient and exact computation and manipulation of algebraic points and curves. The library is a collection of C++ classes which can be used to define, manipulate, and query points and curves in the plane. The library incorporates several new approaches that enable easier manipulation of geometric objects. MAPC uses exact computation and uses several algorithms, including ones based on resultants and Stürm sequences. We present several new algorithms which are used to *speed up* the underlying computations.

Main Contributions: We divide our contributions into system contributions and algorithmic contributions.

*Supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Career Award CCR-9625217, ONR Young Investigator Award (N00014-97-1-0631), Honda, Intel and NSF/ARPA Center for Computer Graphics and Scientific Visualization

†Currently at AT & T Research Labs.

The system contributions of this work involve the implementation of the MAPC library. The primary contributions are:

- An efficient and unified representation for manipulating points where each of their coordinates is defined as either a known rational number or an algebraic number.
- A representation for segments of algebraic plane curves that allows them to be easily stored and manipulated for geometric applications.
- A number of routines for performing queries on these primitives.

To speed up the performance of the overall library, we present *three* new algorithms. These are:

- An algorithm for rapidly finding the sign of a determinant of arbitrary size with multiprecision integer entries using a new floating point filter. (section 4.1)
- An new algorithm for rapidly isolating the intersections of two algebraic plane curves within some region. (section 4.2)
- A new, simple algorithm resolving the topology of an algebraic curve. (section 4.3)

Our current implementation is limited to manipulating points and curves in a plane, though it can be extended to handle objects in 3D. We highlight the performance of library on a number of applications. These include curve intersections, decomposing a curve into monotonic components, sorting points along a curve, and computing arrangements of planar curves. In practice, our root isolation algorithm and sign of determinant algorithm have achieved more than an order of magnitude speedup over earlier implementations.

Paper Organization: The remainder of the paper is organized as follows:

- Section 2 presents previous work related to this library.
- Section 3 discusses the representations and uses of the classes defined in MAPC.
- Section 4 presents three new subalgorithms that are implemented in MAPC.
- Section 5 compares our library with some previously developed libraries.
- Section 6 gives examples of timing results for portions of the code, and examples of applications the library has been applied to.

2 Previous Work

In this section, we will describe some of the previous work that has been done in related areas. Section 5 discusses how our approach differs from these previous approaches.

2.1 Signs of determinants

Many important geometric queries can be expressed as the sign of the determinant of a suitable matrix. The classic algorithm for computing the determinant of an integer matrix is based on the Chinese remainder theorem. A recent treatment may be found in [MC93], and an efficient implementation may be found in the LiDIA library [BBP95].

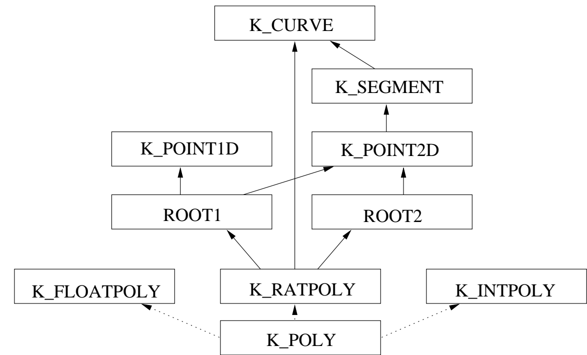


Figure 1: The major classes in MAPC. Dotted lines represent inheritance, and solid lines represent a uses relationship.

Brönniman et. al. [BEPP97] improve this technique with a new Chinese-remainder reconstruction algorithm.

Much of the recent work on the exact determinant-sign problem has focused on small matrices. Karasick et. al. [KLN91] present a variety of techniques based on exact interval arithmetic on matrices of order 2–4. Fortune and Van Wyk [FV93] experiment with a variety of determinant algorithms and filters, and an expression compiler, on matrices of order 3–4. Avnaim et. al. [ABD+94] computes determinant signs of order 2–3 using only single-precision arithmetic, assuming the matrix entries are single-precision.

2.2 2D Root Finding

Finding 2D roots of a pair of bivariate equations is a well studied problem which is usually considered in a more general setting of finding roots of a set of n equations. A number of approaches have been used to solve this. One approach is to use worst-case bit length estimates to guarantee accurate results (e.g. [Can87, Yu92]), Another approach involves the use of multivariate Stürm sequences (e.g. [Ped91, Mil92]). Grobner Basis methods are commonly used, particularly in general computer algebra systems. Finally, a number of other approaches, including those based on interval arithmetic have been explored.

2.3 Algebraic Curve Topology

The curve topology algorithm which we discuss in section 4.3 is closely related to the problem of finding a Cylindrical Algebraic Decomposition (CAD) for a curve, although the CAD is more general. [ACM84a, ACM84b] gives a good overview of the CAD. [AF88] is one example of an approach to compute the CAD. Some work has addressed the specific problem we deal with more directly, notably [KYP92] (although their method is not exact).

3 Representations

In this section we discuss the representations used by the C++ classes in our library. We first give an overview of the representations, and then discuss how these representations are particularly useful in geometric applications. Figure 1 gives a overview of the hierarchy for the major classes provided.

3.1 Overview of Classes

Our library uses the LiDIA ([BBP95]) library to provide exact rational arithmetic. The LiDIA library offers many data types, but we use only the *bigrational* and *bigint* classes (along with some of their matrix classes defined on these). LiDIA's exact rational arithmetic forms the basis for most of the underlying arithmetic operations in our library.

- *K_POLY, K_FLOATPOLY, K_INTPOLY, K_RATPOLY:* The K_POLY class implements a generic multivariate polynomial. From the base K_POLY class, we define K_FLOATPOLY, K_INTPOLY, and K_RATPOLY classes, which have coefficients of type double, bigint, and bigrational, respectively. The polynomials are kept in a dense representation, with an arbitrary number of variables. The space allocated to the coefficients is related to the maximum degree in each variable. For example, a bivariate polynomial with a maximum degree of 2 in x and 3 in y would have space allocated for 12 coefficients. Because the representation is dense, K_POLYs are not appropriate for polynomials with a very high degree or a very large number of variables. Most subsequent classes are based on the use of the K_RATPOLY class.

- *ROOT1:* ROOT1s represent roots of univariate polynomials. The roots are represented as an interval, and roots are isolated using Stürm sequences.

- *ROOT2:* ROOT2s represent roots of a pair of bivariate equations. ROOT2s isolate roots either by the use of a bivariate Stürm sequence (described in [KKM97]) or by a new method outlined in section 4.2. The underlying representation of a ROOT2 may be as either a 2D interval, or (preferably) as a pair of 1D intervals (i.e. two ROOT1s).

- *K_POINT1D:* K_POINT1Ds are used to represent "1D points," although they can be thought of as representing any algebraic number. A K_POINT1D allows a point to be represented as either a known rational number or as the root of a K_RATPOLY within a specified open interval (represented using a ROOT1). This representation allows one to avoid the unnecessary overhead involved whenever the root is known more precisely than "within an interval," but allows both types of points to be stored in a unified form.

- *K_POINT2D:* K_POINT2Ds are used to represent "2D points." A K_POINT2D allows a point to be represented as either a pair of rational numbers, a rational number in one dimension and the root of a K_RATPOLY within an interval in the other, or as a root of two K_RATPOLYs within a 2D interval (represented as a ROOT2). Figure 2 illustrates the cases.

- *K_CURVE, K_SEGMENT:* The K_CURVE class is used to represent curves in the plane. Each K_CURVE consists of an equation, $f(x, y) = 0$, along with a number of K_SEGMENTs. The K_SEGMENT stores only a starting point and an ending point. A string of K_SEGMENTs (where the starting point of one is the ending point of the previous) forms a curve. This representation allows us to work with curves in useful ways, so that we can represent only the portion of a curve that we are interested in, and make computations relative to that portion easily. In addition, one of our new algorithms (see section 4.3) allows us to ensure that any K_SEGMENT defines only a monotonic portion of the curve. Monotonicity assists us in performing a number of queries. Our representation of curves allows them to be used and manipulated readily in a geometric program. Figure 3 gives an example of the way a K_CURVE could be stored.

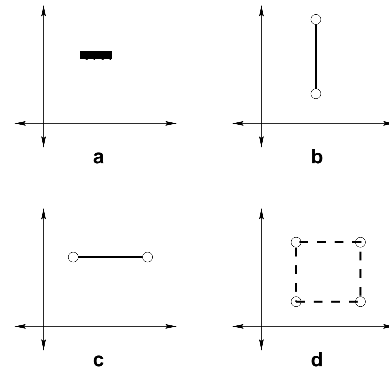


Figure 2: Four ways of representing K_POINT2Ds: a) Both coordinates known as rational numbers b) x known as a rational number, y known as the root of a polynomial within an open-ended interval. c) Same as case b, with x and y reversed. d) As a root within an (open bordered) two dimensional interval.

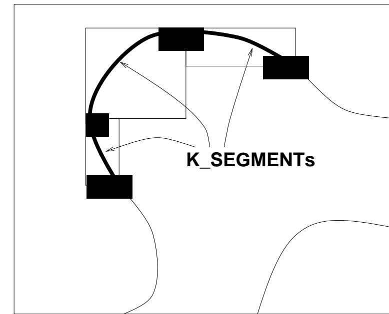


Figure 3: An example of a K_CURVE (the bold portion of the algebraic plane curve). The K_CURVE is represented using three K_SEGMENTS, each of which defines a monotonic portion of the curve. Notice that only the portion of interest is kept—the other component and other segments of the same component are not wanted and thus are not stored.

The classes as defined above are useful for a number of geometric operations. Details of many of the specific operations provided are given in [KCMK98].

3.2 Efficient and Usable Representations

The representations that MAPC provides were designed to provide for both efficiency and ease of use. Some of the specific aspects of MAPC that contribute to its ease of use include:

- A uniform representation for all points, regardless of whether they have rational or algebraic coordinates. All operations on the points automatically use a routine appropriate to the underlying representation.

- Automatically storing points in the simplest form possible, when a simpler form arises in the course of another computation.

- Having a method for storing curves that allows specific segments of the curve to be stored and used.

Some of the aspects of MAPC that contribute to efficiency include:

- Incorporates a new, fast 2D root finding algorithm to handle most curve-curve intersections.

- Makes use of a new, fast sign-of-determinant algorithm to speed up cases still requiring 2D Stürm calculations.

- Makes use of very fast floating-point root estimates to speed up exact root finding procedures.

- Searches for "shortcuts" and special cases where root finding can be performed quickly.

- Maintains a representation for curves which allows several operations (e.g. intersections, sorting points) to be performed quickly.

4 New Algorithms

In this section, we present some new algorithms and their implementation in MAPC. Specifically, the methods described here are approaches for rapidly finding the sign of a determinant, computing the intersections of two algebraic plane curves, and decomposing an algebraic plane curve into monotonic segments over a region.

These algorithms were developed in order to increase the efficiency of the common operations available with the library. The sign of determinant routine is used as a key component in the two dimensional Stürm sequence approach ([KKMC98]) which is included with MAPC. The 2D root isolation algorithm offers a dramatic speedup over the 2D Stürm approach in finding the intersection of curves in the plane, which is a common operation in many applications. The curve decomposition algorithm is used to break a curve down into segments so that it can be easily handled and geometric operations on that curve can be performed quickly. Although these algorithms were developed in the context of efficiency within MAPC, each of them could be applied to problems outside the domain of MAPC.

We discuss how these algorithms are different from previous approaches in section 5

4.1 Determinant Sign Computation

An efficient implementation of multivariate Stürm sequences needs an efficient routine for computing the sign of the determinant of an integer matrix ([KKMC98]). For a typical problem involving the intersection of two degree-four curves, the matrices are on the order of 31×31 with integer entries which are often 100 bits or more.

Although the basic MAPC approach does not rely on multivariate Stürm sequences (as we will see in section 4.2), a Stürm approach may be used as an alternative or to check for certain specific types of intersection. For this reason, we include our bivariate Stürm sequence code, including our determinant sign routine, as part of the MAPC implementation.

The fastest general-purpose algorithm for computing the sign of the determinant of an integer matrix is based on the Chinese Remainder Theorem. First, a bound on the magnitude of the determinant is computed (Hadamard's bound [Knu69]). A number of machine-size primes $p_1, \ldots, p_m$ are chosen such that their product P exceeds twice Hadamard's bound. When the determinant is computed modulo P and interpreted as an integer between $-H$ and H, it is known exactly. To compute the determinant modulo P, it is computed modulo each p_i (typically using Gaussian elimination), and reconstructed from these residues.

Before running the Chinese-remainder code, we use a new filter based approach proposed by Demmel [Dem98]. The integer matrix is approximated with a double-precision floating-point matrix, and its singular value decomposition (SVD) is computed using the LAPACK library [ABB+92]. The determinant sign is easily read off from the SVD. To diagnose the correctness of this sign, we apply a backwards error bound, as described in [DK90]. We use the L_2 matrix norm. The computed SVD of the matrix A is the exact SVD of a nearby matrix A', and the distance $\|A - A'\|$ can be bounded. If $\|A - A'\|$ is smaller than the distance from A' to the set of singular matrices, then the determinants $|A|$ and $|A'|$ have the same sign, and the filter has succeeded.

The bound on $\|A - A'\|$ and the distance from A' to the set of singular matrices can both be computed from the SVD. The bound is

$$\|A - A'\| \leq \sigma_1 \cdot f(n) \cdot \epsilon,$$

where σ_1 is the largest singular value of A, $f(n)$ is a function of the size of the matrix, and ϵ is machine epsilon. The distance to the set of singular matrices is just σ_n, the smallest singular value. So our filter succeeds if

$$\sigma_n \geq \sigma_1 \cdot f(n) \cdot \epsilon.$$

This can be rewritten in terms of the matrix condition number $\kappa = \sigma_1/\sigma_n$:

$$\kappa \leq 1/(f(n) \cdot \epsilon),$$

revealing that the matrices that fail the filter are exactly those which are considered ill-conditioned. Notice that we need $\sigma_1(A)$ for the error bounds, but we use instead $\sigma_1(A')$—this error is ignored.

The function $f(n)$ is known to be at worst $100n^3$ [Dem98]. (We use $f(n) = 100n^3 + 1$ to account for the rounding error in representing A as a floating-point matrix.) The LAPACK User's Guide [ABB+92] suggests that when computing backwards error bounds on the SVD, $f(n) = 1$ is realistic. We have verified this experimentally: on 2000 randomly-generated Sylvester matrices of size 31×31, the assumption $f(n) = 1$ caused no false positives (that is, the filter either computed the correct sign or reported "no confidence").

In summary, we compute determinant signs with a three-stage filter:

- Compute the SVD. Stop if the sign is known.

- If all entries have 53 bits or fewer, use Sylvain Pion's implementation of the algorithm in [BEPP97].

- Otherwise, apply the full Chinese-remainder determinant as implemented in LiDIA.

4.2 2D Root Isolation

A key element in our library is determining the intersection points of two algebraic plane curves within a region. We need to find a number of two dimensional intervals, each

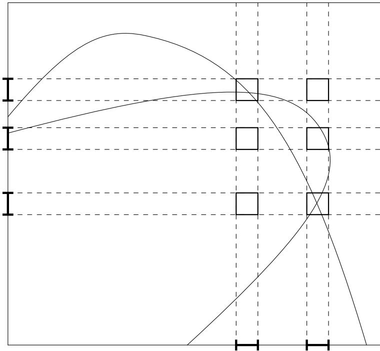


Figure 4: Initial step of 2D root isolation.

of which is guaranteed to contain exactly one intersection point of the two curves.

Given two curves, $f(x,y)=0$ and $g(x,y)=0$, along with a domain, $x=[x_1,x_2]$, $y=[y_1,y_2]$, we need to isolate all of the real intersections of the two curves within the domain.

- First we compute $X(x) = Res_y(f,g)$ and $Y(y) = Res_x(f,g)$, where Res_i refers to the resultant of the two polynomials eliminating i. The real roots of X and Y are the x and y coordinates, respectively, of the common real solutions of $f=0$ and $g=0$.
- We isolate the roots of $X=0$ in the range $x=[x_1,x_2]$, and $Y=0$ for $y=[y_1,y_2]$, using an exact univariate root finding approach (e.g. as in [KKMC98]).
- If there are m roots of $X=0$ in the region, and n roots of $Y=0$ in the region, then we form mn 2D "boxes" each of which may or may not contain a common real intersection of $f=0$ and $g=0$. The boxes are formed from the intersection of a root interval from $X=0$ and one from $Y=0$.

  An example is shown in Figure 4. In the figure, the two roots of $X=0$ and three roots of $Y=0$ are shown along the x and y axes. The boxes (in this case, 6 of them) in which there may be roots are highlighted.
- For each of the roots of $X=0$ (and similarly for $Y=0$), we have a lower and upper bound, say a_1 and a_2. We find all roots of $f(a_1,y)$, and determine which (if any) lie on the boundaries of one of the n boxes which has a_1 as a lower boundary. We refer to these intersections as *box hits*, since they are where the curve "hits" the box. This is done similarly for $g(a_1,y)$, and then repeated for the upper bound, a_2. In total, there will be $2(m+n)$ univariate equations for which roots must be found. Referring to figure 4, this means finding the intersections of the two curves with the dashed lines.
- We must determine which (if any) of the boxes actually contain a common root. Note that there can be at most one root inside any box. The fundamental observation is that we can order the box hits around the box and determine whether there is an intersection within the box. The details of how to determine whether or not a box has a root inside of it are discussed in [KCMK98].

We now briefly mention a few considerations which may also be taken into account. If one wishes to find all intersections between the two curves, then a bound on the maximum and minimum sizes of real roots of $X(x)=0$ (and $Y(y)=0$) can be obtained (e.g. as in [Dav93]). This gives bounds in x and y for the real intersections of $f=0$ with $g=0$, which thus defines the test region. Also, if one has some kind of a priori knowledge of the number of intersections in the region (e.g. by taking the bound on maximum number of intersections as a function of the polynomial degrees), then one can use an approximate method (such as a floating-point based solver) to find initial boxes to test. Also, if it is guaranteed that any root of $X(x)=0$ (or $Y(y)=0$) corresponds to only one intersection between the two curves, then it is possible to limit the number of box boundaries which need to be tested.

4.3 Curve Topology

In order to be able to manipulate and effectively use curves in our library, it is necessary for them to be decomposed into monotonic segments. The specific problem faced is this: given a polynomial, $f(x,y)=0$, and a domain, $[x_1,x_2]$, $[y_1,y_2]$, decompose the curve into monotonic segments of $f=0$ within the domain, and record the connectivity between monotonic segments. By "monotonic segments," we mean branches of the curve, delimited by two points which lie on the curve, such that as one traverses the curve from one endpoint to the other, the curve is non-increasing (or non-decreasing) in both x and y. We refer to this problem as resolving the *curve topology*.

Before running our algorithm, we must isolate all of the turning points (local maxima and minima) within the domain of interest. The turning points are the intersections of $f=0$ with each of its two partial derivative curves, $f_x=0$ and $f_y=0$. We find the turning points by isolating these intersections, using the method described in section 4.2; any method will suffice, but the technique described below benefits from the planar subdivision constructed during root isolation. Special care must be taken for singular points, which we define as common solutions of $f=0$, $f_x=0$, and $f_y=0$. We classify these points into three categories. First are point components of $f=0$. The algorithm will ignore the point component. (Full 2D Stürm sequences can be used to detect this case if necessary.) Second are cusp points, where the curve comes to a "point." Cusp points should be treated the same way as turning points in the algorithm below. However, it is necessary to make sure that the cusp point is treated as only a single point—not as two turning points. The last type is the self-intersection, where $f=0$ "crosses over" itself. We discuss how self-intersections can be incorporated into this algorithm in [KCMK98]. The only real difficulty arises in that it is not clear how to represent the connectivity between the multiple curve segments which connect at a self-intersection. Currently MAPC cannot represent curves with multiple branches. These three types of singularities must be identified and either handled or avoided separately prior to running this algorithm.

We also compute *edge points*, the intersections of $f(x,y)=0$ with the boundaries of the region of interest. Edge points are found as the roots of $f(x_1,y)=0$, etc. Our approach uses a recursive subdivision of the region of interest to find the connectivity between all turning points and edge points, thus computing a decomposition of the curve into monotonic regions. Each subdivision involves taking a vertical or horizontal line and finding all intersections of the curve with

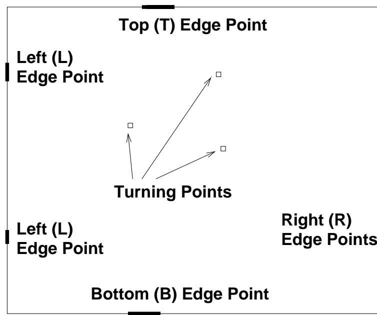


Figure 5: An example curve topology algorithm input.

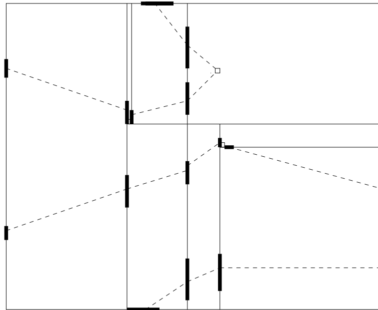


Figure 6: The results of the curve topology algorithm. The dark bars represent points found on the curve during intermediate steps, and the dashed lines represent the connection between the points. The overall curve has 3 components in this region.

that line. These intersection points are then edge points in the two subregions.

Figure 5 shows an example of a starting configuration for our routine. The turning points have all been isolated, as have the edge points. The edge points are further classified by edge.

The algorithm proceeds by analyzing the pattern of turning points and edge points in the region, and either finding the connections between them, or subdividing the region. The details of how the subdivision or connections are made are given in [KCMK98]. When the algorithm completes, the curve has been divided up into monotonic segments.

Figure 6 shows the results of our algorithm for the example case presented in figure 5. The connectivity between all edge points and turning points has been found, thus breaking down the polynomial into a number of monotonic segments. The figure shows the subdivided regions, along with the intermediate edge points computed (exaggerated in the drawing for clarity) by our algorithm. True examples and timings are presented in section 6.3.

Each monotonic subsection can be considered to have a "bounding box" surrounding it. We can further subdivide the curve so that the non-adjacent subsegments have non-overlapping bounding boxes. This allows us to determine what portion of the curve a point lies on simply by finding which bounding box it is in. A direct way of performing such a subdivision has been discussed in [KKM97].

5 Comparisons with other systems

In this section we compare how MAPC differs from previous libraries. We are not aware of any library which allows the unified exact representations for points and curves that MAPC has. We mention a few related libraries, however, and how they differ from ours.

- *LiDIA* [BBP95] offers, among other things, routines for manipulating exact multiprecision integer and rational numbers, as well as polynomials. We have used LiDIA's rational number routines as the basis of our implementation, but use our own polynomial classes to allow more specialized (and more efficient) manipulations geared to low-degree geometric problems. LiDIA is not designed for (and does not offer) any geometric data structures of its own.
- *LEDA* [MN89] offers an extensive set of routines for dealing with geometric problems. It includes routines such as those in LiDIA for rational numbers, as well as routines for representing 2D points with floating point or rational coordinates, and performing queries on them. It also includes a way of representing real algebraic numbers by allowing numbers to be represented as the nth root of another number. LEDA does not contain any support for manipulating (non-linear) curves in the plane, however. Moreover, points with algebraic coordinates which are defined as the intersection of such curves cannot be stored or manipulated.
- *CGAL* [FGK+96], like LEDA, offers a framework for performing a number of geometric queries and makes use of LEDA's classes. It offers a wider range of geometric objects and algorithms than LEDA, but the key addition is the use of a template format to allow for extensibility to other representations. Like LEDA, there is no direct support for algebraically defined points. The template format, however, offers the possibility that points as handled by MAPC could be included under the CGAL framework. There would need to be some significant additions and modifications to allow the curve structures used in MAPC to be handled under the current CGAL approach, however.
- *APU* [Reg96] provides a number of utilities for solving algebraic problems involving polynomials. The functions provided by APU are similar to many of the underlying functions provided on K_RATPOLYs in MAPC, along with much of the functionality used to find and isolate roots. APU, however, does not define any geometric data structures.
- *FRISCO and POSSO* provide routines for solving polynomial systems. Thus, they are capable of performing some of the computations necessary for determining points. However, they do not provide any of the geometric functionality included in MAPC. Also, these programs are geared toward more general systems than our approach, which deals only with intersections of planar curves. By specializing our approach in this way, we are able to achieve greater efficiency than we would otherwise.

6 Example Applications

In this section we present some timings for the three new algorithms we have presented, as well as a couple of ap-

plications. All timings are in CPU seconds on a 400 Mhz Pentium II processor running Linux.

6.1 Sign of Determinant Results

We evaluated our filter and its stages using several test groups of 100 random matrices. All of the matrices are 31×31 submatrices of the Sylvester resultant of a polynomial $f(s)$ with its derivative $f'(s)$. Each matrix entry is 0 or a coefficient of either f or f'. Such a matrix is typical of the largest matrices encountered in a two-dimensional Stürm computation examining the intersections of two degree-four curves. The coefficients of f are random numbers.

We use two random number generators. The "b-generator" chooses coefficients of f so that the coefficients of f' (and thus all matrix entries) are at most b bits. With this strategy, the Sylvester matrices tend to be well-conditioned. In our experiments, we used $b = 16, 53, 128$.

The "λ-generator" produces ill-conditioned matrices that exercise our filter. A constant $0 < \lambda < 1$ is chosen. Each coefficient of f is constructed as follows. Initialize the coefficient to a random 16-bit number. Choose a random number r uniformly from $[0, 1]$. If $r \leq \lambda$, concatenate another 16 random bits to the coefficient and repeat; otherwise, the coefficient is complete. In other words, the probability that the coefficient consists of at least $16m$ random bits is λ^{m-1}. For our experiments, we generate 100-matrix test groups for $\lambda = .1, .2, .3$; entries were no larger than 65, 82, and 116 bits, respectively.

Comparison of methods. In figure 7, we show that the three-stage filter improves on the speed of the general Chinese-remainder algorithm as implemented in LiDIA. The routine we call "Inria" is Sylvain Pion's implementation of the algorithm in [BEPP97]. The Inria code and LiDIA implement essentially the same algorithm, and both take advantage of IEEE double-precision floating-point to compute in modular arithmetic. The Inria code allows as input only matrices with entries up to 53 bits in length, the largest integers that will fit in a `double`. For this reason, the Inria code cannot handle some of the λ tests in figure 7, which may have more than 53 bits. LiDIA allows matrix entries of arbitrary size, using the library's `bigint` datatype.

For our large matrices, most of the time is spent in computing the determinant modulo the various primes. The reconstruction step is relatively cheap. Thus, the Inria code is faster than LiDIA mainly because it uses only machine-size numbers. The two routines use essentially the same algorithm—the main difference is LiDIA's use of `bigint` matrices during modular reduction and determinant reconstruction. Comparing the speed of these two implementations reveals the amount of overhead incurred in using `bigints` at all. Our results suggest that if we could remove *all* memory allocation problems (calls to `malloc()`, matrices spread across multiple memory pages, and so forth) from the determinant routine, we could attain a speedup of as much as 12. (Some techniques for improving the behavior of `bigints` with respect to memory management are explored in [FV93].)

Utility of filter stages. Figure 8 shows the three stages in our filter in order, together with the number of matrices (out of 100) which terminate at that stage. This demonstrates the efficacy of the various stages. It also shows that the filter has very different behavior on different families of matrices.

Time spent in filter stages. We show the percentage of time spent in each of the three filter stages in figure 9.

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
LiDIA	13.3	44.4	110.1	24.3	30.4	38.0
Inria	1.4	3.7	—	—	—	—
Filter	0.5	0.7	0.8	9.7	24.2	38.5

Figure 7: Comparing the speed of the filter with other exact methods (which are both used as subroutines in the filter). Times are in seconds.

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
SVD	100	100	100	42	18	7
Inria	—	—	—	33	17	2
LiDIA	—	—	—	25	65	91

Figure 8: Number of matrices "caught" by each stage of the filter.

6.2 Isolating Roots

Figure 10 gives some example timing results for our new root isolation algorithm. The algorithm isolates all roots of two bivariate equations to a specified precision. The table lists the degrees of the two curves, the number of bits in the coefficients of the two curves, the number of actual roots in the region of interest, the time taken by an earlier multivariate Stürm sequence based approach (for comparison), the time taken by our new approach, and a percentage breakdown of the time spent in the three major stages of our approach. The three major stages listed are the resultant computations for eliminating one variable out of the initial equations, the time to isolate the *initial* univariate roots, and the time to generate all the "box hits" (includes several univariate root finding steps). All isolations find the roots to an interval of no larger than 0.001 of the original domain.

Notice that as the cases become more complex, the time spent finding the initial 1D roots (but *not* the 1D "box hit" roots) begins to dominate the overall computation time. This is primarily due to the univariate polynomials having a high degree and large coefficients. If the initial polynomials have degrees m and n, and the coefficients have bit lengths a and b, then the univariate polynomials being dealt with have degree mn and coefficient bit length $(an + bm)\log_2(m + n)$.

6.3 Curve topology

Figure 11 presents some example timing results from our curve topology algorithm, along with the degree of the equation, the number of bits in the coefficients of the equation, the number of turning points and different components of the curve *in the region of interest*, the time taken to isolate all the turning points, and the time taken to run the topology resolution algorithm itself. The figures show the curves, along with the points found on the curve (including turning points) during the topology computation algorithm. The points on the curve are connected in order with straight lines. No removal of overlapping bounding boxes is performed. All points found on the curve, including both

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
SVD	100%	100%	100%	3%	4%	4%
Inria	—	—	—	10%	11%	13%
LiDIA	—	—	—	87%	85%	83%

Figure 9: Time spent in each stage of the filter.

	1	2	3	4	5
Degree of Curves	2, 2	2, 4	3, 3	4, 3	4, 4
Bits in Coefficients	3, 6	9, 24	23, 20	18, 23	24, 18
Number of Roots	4	2	2	2	1
Time using 2D Stürm	0.51	8.21	20.26	123.29	333.48
Time Using New Algorithm	0.07	0.28	1.01	8.97	36.92
% of Time in Resultants	16	43	17	4	2
% of Time for 1D Roots	30	42	79	95	98
% of Time to Find Box Hits	54	15	5	1	0

Figure 10: Timing Results (in seconds) for Root Isolation Algorithm. Five test cases are shown for pairs of curves of varying degree and coefficient size, Timings are presented using both a heavily optimized 2D Stürm algorithm and our new algorithm. The time spent in the three main portions of the new routine is given in percentages.

turning points and points found by subdivision in the intermediate steps, are shown.

Notice that the algorithm runs quite fast, and its running time is almost negligible in comparison to the time for computing turning points. Computing the turning points, then, is the bottleneck.

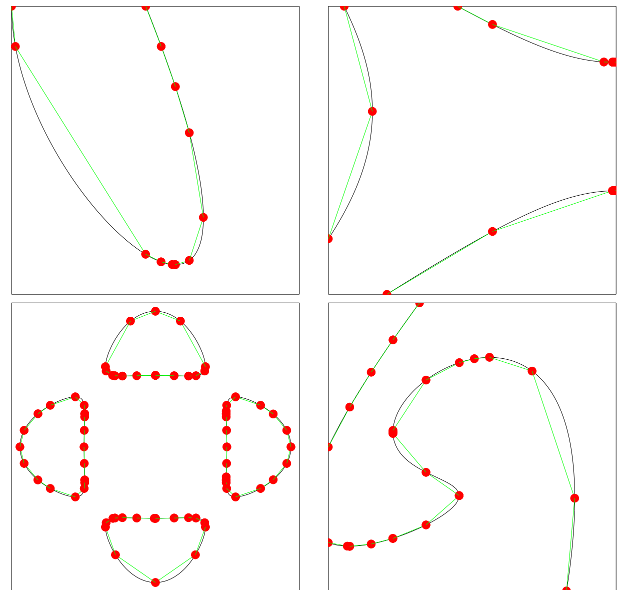
6.4 Sorting points along a curve

One application for MAPC is the problem of sorting points along a curve. This problem is a key step in the computation of the medial axis of a polyhedron [CKM98]. Given the representation provided in MAPC the sorting procedure is simple. Once the curve has been broken into monotonic segments with non-overlapping bounding boxes, we simply find which of the bounding boxes contains each of the points known to be on the curve. Within any one bounding box, the points can be sorted by either x or y, and the bounding boxes themselves are already sorted along the curve as part of MAPC's representation.

Figure 12 shows one example of a curve and a group of points which have been sorted along it. The points were generated by intersecting the curve with 25 curves of varying degree and coefficient bit length, resulting in 52 points on the curve. The total time taken to generate the points (25 curve-curve intersections) was 102.3 seconds. The time taken to resolve curve topology and to sort the points along the curve was less than 1 second.

6.5 Arrangement of planar algebraic curves

We have implemented an algorithm which computes the faces generated by an arrangement of planar algebraic curves inside a specified rectangular box. Treating the boundary, turning, and intersection points as vertices and the curve segments as edges, we construct a homeomorphic planar straight-line graph. Figure 13 illustrates one such arrangement. The algorithm proceeds in three steps. Initially, all the curves are clipped to the boundary rectangle and parsed into monotonic segments. All the pairs of curves are then



	1	2	3	4
Degree of Equation	3	4	4	5
Bits in Coefficients	20	18	5	60
Number of Turning Points	2	3	24	5
Number of Components	1	3	4	2
Time to Find Turning Points	0.15	0.48	0.85	92.27
Time to Run Algorithm	0.04	0.06	0.21	0.09

Figure 11: Curve Topology Algorithm Results and Timings. The figures show four test cases (1 and 2 on top row, 3 and 4 on bottom), along with the points which the curve topology algorithm finds, connected by lines. The table gives information about each curve within the domain, the time required by the algorithm to isolate the turning points, and the time required (in seconds) to run the topology algorithm.

tested for intersection, and the intersection points are computed. The final step of the algorithm uses a simple anticlockwise ordering of all the edges around a vertex to systematically read out all the faces. Figure 13 also shows timings for the example cases.

7 Conclusion and Future Work

We have presented a description of MAPC, our library for exact representation of points and curves in the plane. The library allows us to easily define points (in multiple formats) and curves in the plane, and manipulate and perform queries on these objects. We have also presented three new algorithms that have been implemented as a part of MAPC, and have provided some timing results and examples of problems MAPC can be used for.

Further information about MAPC, including its source code, is available at http://www.cs.unc.edu/~geom/MAPC/.

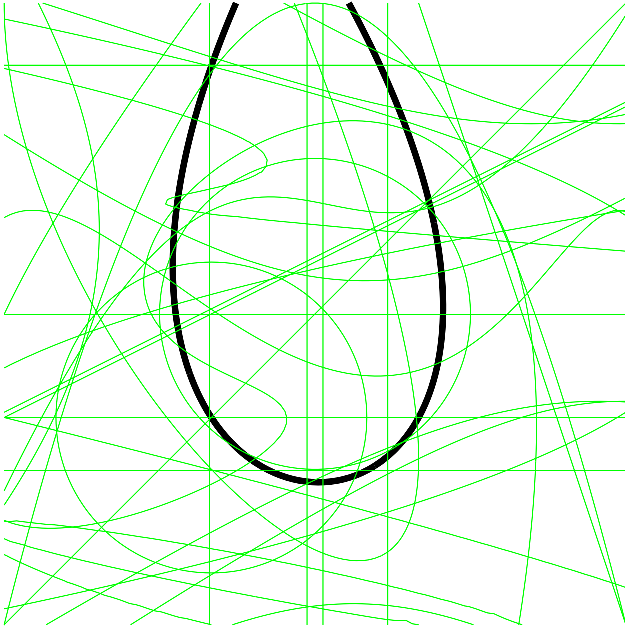
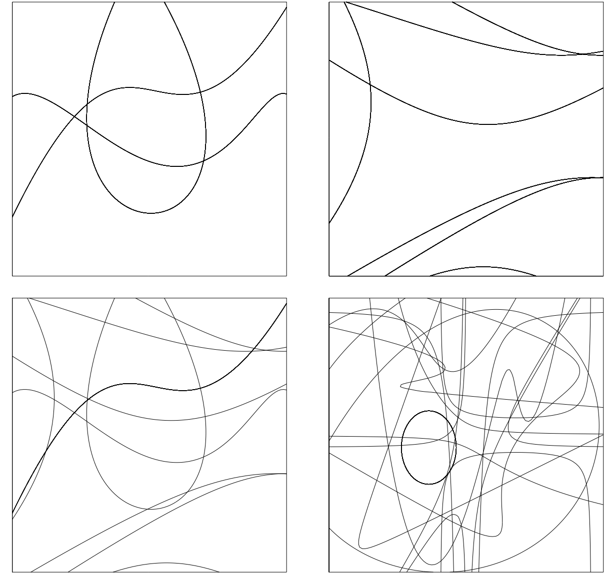


Figure 12: Sorting Points Along a Curve. The curve along which the points are sorted is in bold. The points to be sorted are the 52 intersections of the bold curve with the other 25 curves shown. Finding all intersections takes 102.3 seconds, and the time to perform curve topology on the bold curve and sort the points takes less than one second.



Case	1	2	3	4
Number of Curves	3	3	6	12
Coeff. Bit size (Num./Den.)	25/1	19/14	25/14	62/17
Number of Faces	9	11	31	171
Time (in secs.)	8.38	16.95	120.89	1142.21

Figure 13: Arrangement of planar algebraic curves. The figures show a region partitioned into a number of faces by the arrangement of curves. The application finds all subregions, the segments of curves bounding each subregion, and hte connectivity between subregions. Cases 1 and 2 are on the top row, 3 and 4 on the bottom. The table shows the bitlength of the coefficients of the curves (numerator bitlength, denominator bitlength), the number of faces generated by the arragement, and the total time (in seconds) taken to compute the arrangement. The curves have maximum degree 4.

MAPC is already being used as a component in a solid modeling system [KKM97] and an implementation of a polygonal medial axis algorithm [CKM98]. Future development of the MAPC library itself may include:

- Definition of a standard API which would be useful for any operation desired on 2D points and curves. Although MAPC is a step in this direction, we do not claim to have defined a complete API.
- Extending it to use 3D points, curves, and surfaces
- Extending the functionality of the K_POINT2D class to handle other possible representations of points.
- Allowing other, more compact curve representations when curves fall into simpler categories.

7.1 Acknowledgements

We would like to thank Jim Demmel for his suggestion of the floating-point filter used in our determinant sign computation.

References

[AB88] S.S. Abhyankar and C. Bajaj. Computations with algebraic curves. In *Lecture Notes in Computer Science*, volume 358, pages 279–284. Springer Verlag, 1988.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. *LAPACK User's Guide, Release 1.0.* SIAM, Philadelphia, 1992.

[ABD+94] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. Research Report 2306, INRIA, BP93, 06902 Sophia-Antipolis, France, 1994.

[ACM84a] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM J. Comput.*, 13(4):865–877, 1984.

[ACM84b] D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition II: The ad-

jacency algorithm for the plane. *SIAM J. Comput.*, 13(4):878–889, 1984.

[AF88] S. Arnborg and H. Feng. Algebraic decomposition of regular curves. *Journal of Symbolic Computation*, 5:131–140, 1988.

[BBP95] I. Biehl, J. Buchmann, and T. Papanikolaou. Lidia: A library for computational number theory. Technical Report SFB 124-C1, Fachbereich Informatik, Universitt des Saarlandes, 1995.

[BEPP97] H. Bronnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[Can87] J. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1987.

[CKM98] T. Culver, J. Keyser, and D. Manocha. Accurate computation of the medial axis of a polyhedron. Technical Report TR98-034, Department of Computer Science, University of North Carolina, 1998. To appear: Proceedings of ACM Solid Modeling 99.

[Dav93] J. H. Davenport. *Computer Algebra Systems and algorithms for algebraic computation.* Academic Press, London, 2 edition, 1993.

[Dem98] J. Demmel. Private Communication, 1998.

[DK90] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Computing*, 11:873–992, 1990.

[FGK+96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[For95] Steven Fortune. Voronoi diagrams and delaunay triangulations. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 225–265. World Scientific Press, Singapore, 1995.

[FV93] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[KCMK98] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. Technical Report TR98-038, University of North Carolina, Chapel Hill, 1998.

[KKM97] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate b-rep generation of low degree sculptured solids using exact arithmetic. In *ACM/SIGGRAPH Symposium on Solid Modeling*, pages 42–55, 1997.

[KKMC98] J. Keyser, S. Krishnan, D. Manocha, and T. Culver. Efficient and reliable computation with algebraic numbers for geometric algorithms. Technical Report TR98-012, Department of Computer Science, University of North Carolina, 1998.

[KLN91] Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[Knu69] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1969.

[KYP92] David J. Kriegman, Erliang Yeh, and Jean Ponce. Convex hulls of algebraic curves. In J. D. Warren, editor, *Proceedings of the International Society for Optical Engineering Volume 1830, Curves and Surfaces in Computer Vision and Graphics III*, pages 118–127. SPIE, Boston, 1992.

[MC93] D. Manocha and J.F. Canny. Multipolynomial resultant algorithms. *Journal of Symbolic Computation*, 15(2):99–122, 1993.

[Mil92] P. S. Milne. On the solutions of a set of polynomial equations. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 89–102, 1992.

[MN89] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Fachber. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1989.

[Ped91] P. Pedersen. Multivariate sturm theory. In *Proceedings of AAECC*, pages 318–332. Springer-Verlag, 1991.

[Reg96] Ashu Rege. *A Toolkit for Algebra and Geometry.* Ph.D. dissertation, Univ. of California at Berkeley, Berkeley, California, 1996.

[She97] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[SP95] E. C. Sherbrooke and N. M. Patrikalakis. Computation of medial axis transforms of 3d polyhedra. In J. R. Rossignac and C. M. Hoffmann, editors, *Proc. Third ACM Solid Modeling Conference*, 1995.

[Yu92] J. Yu. *Exact arithmetic solid modeling.* PhD thesis, Purdue University, 1992.