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

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 of spline 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.

*Currently at AT & T Research Labs

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 when trying to represent curves. While line segments can be represented using only two endpoints, far more information is necessary to represent a curve segment. 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. Finding a way to effectively deal 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], recognizing curved objects from images using aspect graphs [KP90], robot motion planning [Can88], etc. 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 on 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 Sturm 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.

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 when their coordinates are defined as either a known rational number or an algebraic number.
- A representation for portions 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 and exactly finding the sign of a determinant of arbitrary size with multiprecision integer entries using floating point filters. (4.1)
- An algorithm for rapidly isolating the intersections of two algebraic plane curves within some region. (4.2)
- A simple algorithm resolving the topology of an algebraic curve. (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 which 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 Geometric and Algebraic Libraries

We mention here some of the previously developed geometric and algebraic libraries. In section 5, we discuss the differences between these systems and MAPC. LiDIA [BBP95] is a library developed for computational number theory which provides basic rational and polynomial operations that are useful in a number of geometric algorithms. LEDA [MN89] provides a number of geometric data types and geometric algorithms in addition to various number classes. The CGAL library [FGK+96] provides similar support, but in a more extensible format. The APU toolkit [Reg96], much like the work of the FRISCO and POSSO projects, provides support for algebraic numbers.

2.2 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]. 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.

2.3 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 n-variate 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. [Can88, Yu92]), Another approach involves the use of multivariate Sturm 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.

2.4 Algebraic Curve Topology

The curve decomposition 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] gives one example of an approach taken for computing 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. We discuss each class in detail below.

3.1 Overview of Classes

Our library uses the LiDIA ([BBP95]) library to provide exact rational arithmetic. The LiDIA library offers a number of 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

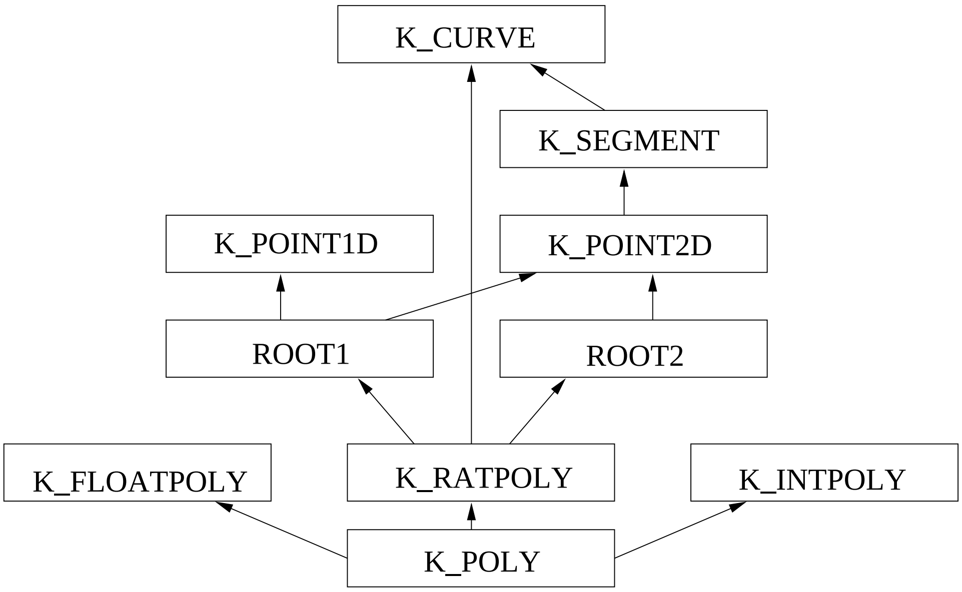


Figure 1: The major classes in MAPC

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. Obviously, since a dense representation is used, 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 allow the representation of roots of univariate polynomials. The roots are represented as an interval, and roots are isolated using Sturm sequences,

- *ROOT2:* ROOT2s allow the representation of roots of a pair of bivariate equations. ROOT2s isolate roots either by the use of a bivariate Sturm 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 1-D 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 format.

- *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). See figure 2 for an illustration.

- *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 with ease. 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 our 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.

The classes as defined above are useful for a number of geometric operations. Details of many of the specific operations provided are given in the appendix, section 8.1.

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:

- Having 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.

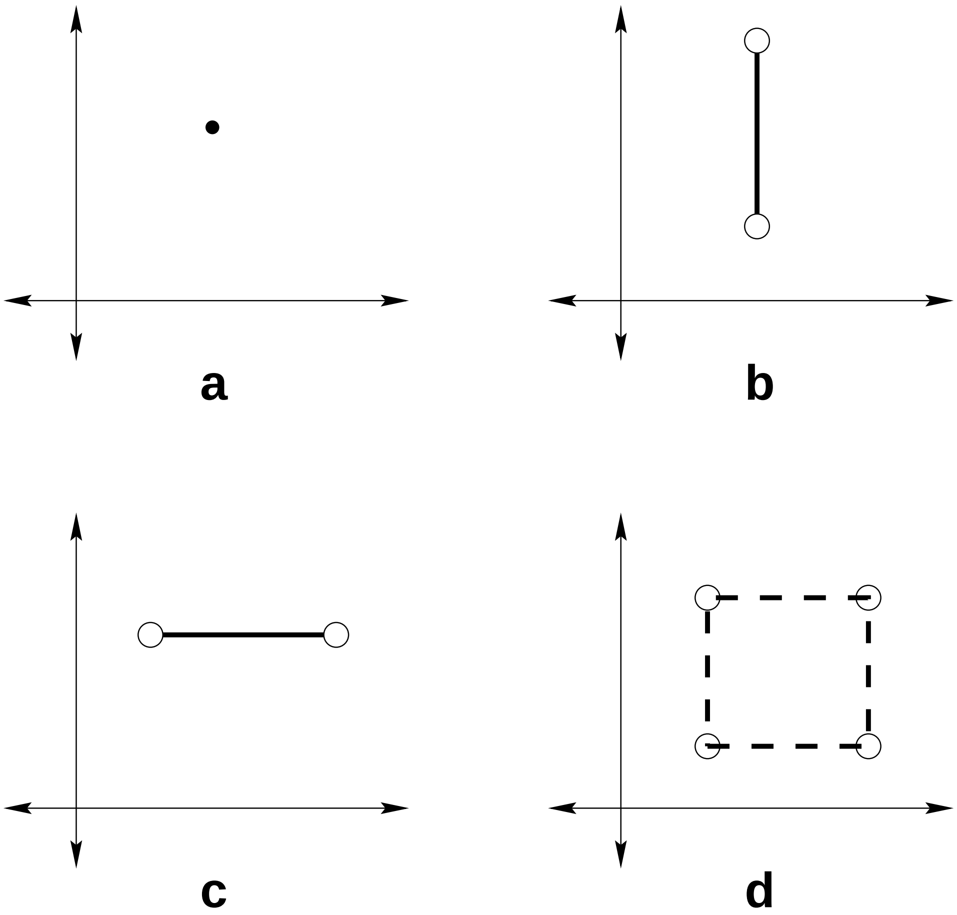


Figure 2: Four ways of representing K_POINT2Ds: a) Both coordinates known as rational numbers b) x known as a rational number, y known as the root of a polynomial within an open-ended interval. c) Same as case b, with x and y reversed. d) As a root within an (open bordered) two dimensional interval.

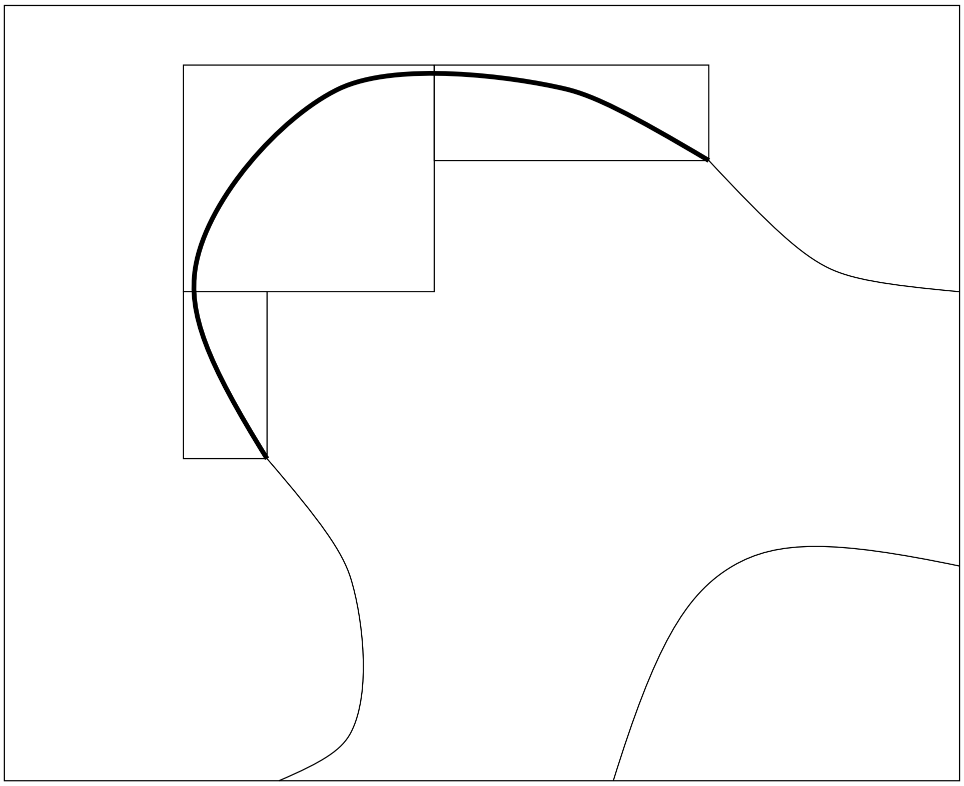


Figure 3: An example of a K_CURVE (the bold portion of the plane algebraic curve). The K_CURVE is represented using three K_SEGMENTS, each of which defines a monotonic portion of the curve. Notice that only the portion of interest is kept – the other component and other portions of the same component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

- 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 portions 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 handle cases still requiring 2D Sturm calculations.

- Makes use of very fast floating-point root estimates to guide 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 involving them (e.g. intersections, sorting points) to be performed far more quickly than otherwise.

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, rapidly finding the intersections of two algebraic plane curves, and finding a decomposition of an algebraic plane curve into monotonic sections 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 Sturm sequence approach which is included with MAPC. The 2D root isolation algorithm offers a dramatic speedup over the 2D Sturm approach in finding the intersection of curves in the plane, which is a common operation in most of the applications we have explored. The curve decomposition algorithm is used to break a curve down into sections so that it can be easily handled and future 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 exactly the sign of the determinant of an integer matrix. For a typical problem involving the intersection of two degree-four curves, the matrices are on the order of 30×30 with entries of 100 bits or more.

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). 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 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 $|A|$ 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 $100n^3$. (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").

The floating-point filter cannot detect zero-determinant cases. Therefore it is preceded by a computation of the determinant over a single prime. If the determinant is nonzero modulo some prime p, then it is nonzero, and we proceed to the floating-point filter. If it is zero modulo p, then the determinant is very likely zero, and we skip to the full Chinese-remainder determinant. Our implementation of this "zero-filter" is simply the first iteration of the Chinese remainder code, modified from LiDIA.

4.2 2-D 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 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 form $X(x) = Res_y(f, g)$ 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, then, are the x and y coordinates 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.

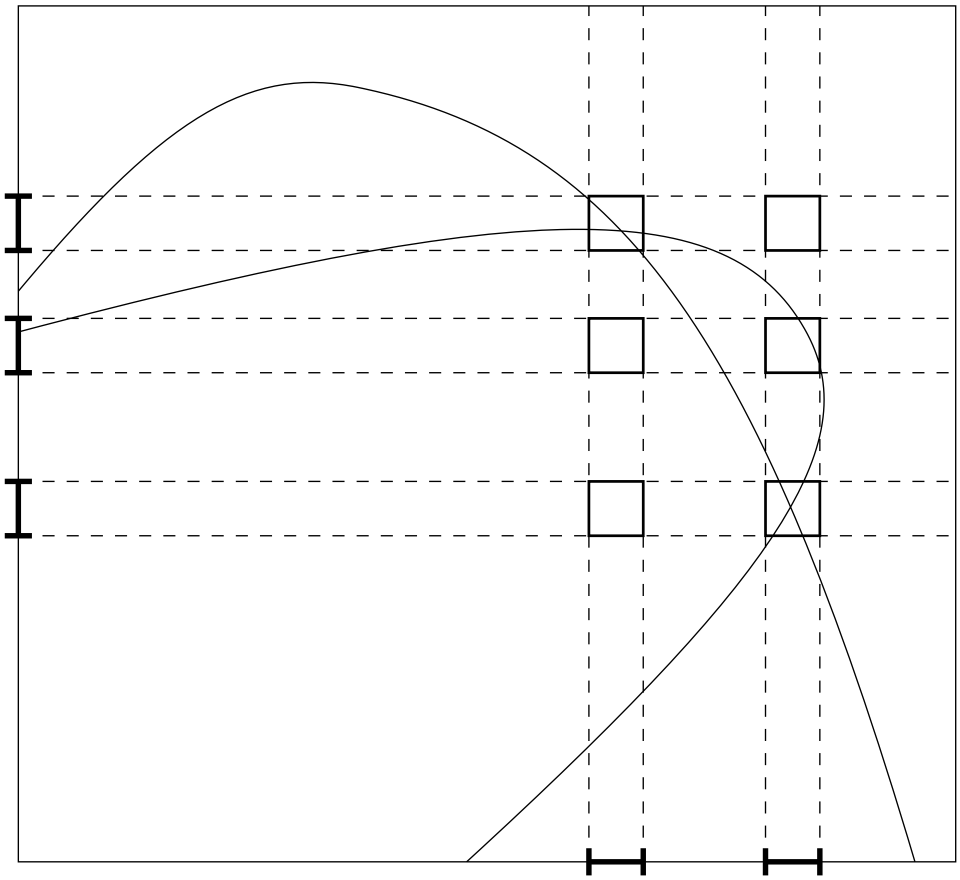


Figure 4: Initial portion of 2-D root isolation.

- If there are m roots of $X = 0$ in the region, and n roots of $Y = 0$ in the region, then we form mn 2-D "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. 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 intersections of the two curves with the box boundaries 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 is discussed in the appendix, section 8.2.

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 $Y(y) = 0$), and thus of the real intersections of $f = 0$ and $g = 0$, can be obtained (e.g. as in [Dav93]), and this used to define 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 determine initial boxes to test. Also, if one can be guaranteed that any root of $X(x) = 0$ (or $Y(y) = 0$) corresponds to only one intersection between the two curves, then you may be able 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 broken up into monotonic segments. The specific problem faced is this: given a polynomial, $f(x, y) = 0$, and a domain, $x = [x_1, x_2]$, $y = [y_1, y_2]$, decompose the curve into monotonic sections of that polynomial within the domain, obtaining the connectivity between monotonic sections. By "monotonic sections", 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 nonincreasing (or nondecreasing) 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. This can be done by finding common solutions between $f(x, y) = 0$ and either $f_x(x, y) = 0$ or $f_y(x, y) = 0$, where the subscript means the partial derivative with respect to that variable.

Although our implementation does this by using the method described in 4.2, these turning points can be obtained by any other method desired. It should be noted, though, that some efficiency gain could be realized if the subdivision of the plane obtained in root isolation were used as the starting point for the following approach.

Special care must be taken for singular points, which we define as places where the $f = 0$, $f_x = 0$, and $f_y = 0$ have a common solution. We classify these points into three categories. First are point solutions. These are cases (as mentioned in 4.2) where the solution of $f = 0$ is only a single point. In this case the algorithm will not automatically identify this point as a component - that must be done separately. Secondly 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. Finally are self-intersections, where $f = 0$ "crosses over" itself. We will discuss the handling of self-intersections at the end of this section. These three types of singularities must be identified and handled separately prior to running this algorithm.

We also compute all intersections of $f(x, y) = 0$ with the boundaries of the region of interest, i.e. the roots of $f(x_1, y) = 0$, etc. These are referred to as "edge points." Our approach uses a recursive subdivision of the region of interest to find the connectivity between all turning points and edge points, thus giving 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 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 depending on which border they hit.

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 the appendix, section 8.3. After the algorithm has been run, the curve has been divided up into a number of 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 6.3.

Each monotonic subsection can be considered to have a "bounding box" sur-

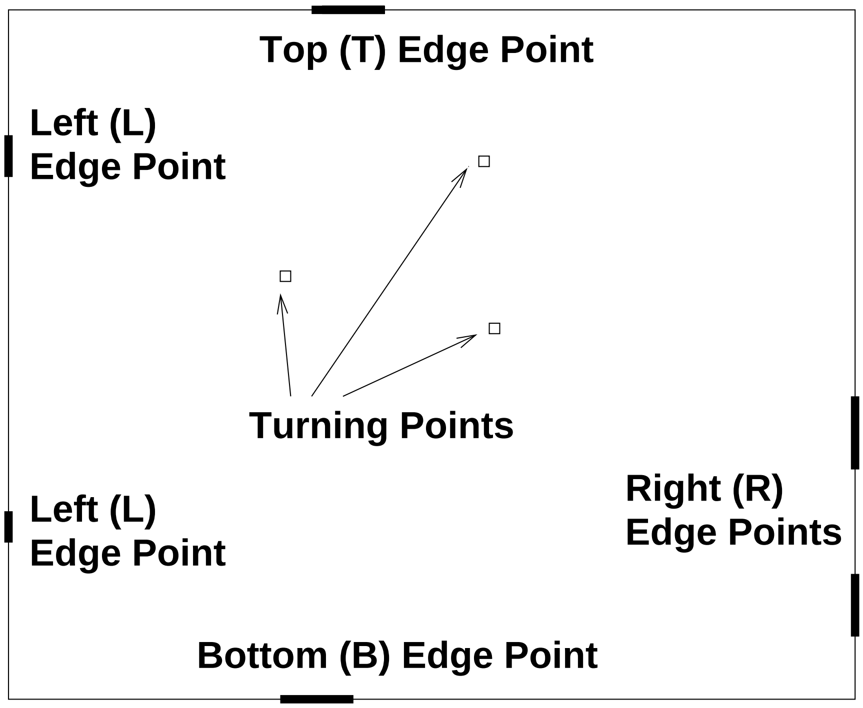


Figure 5: An example curve topology algorithm input.

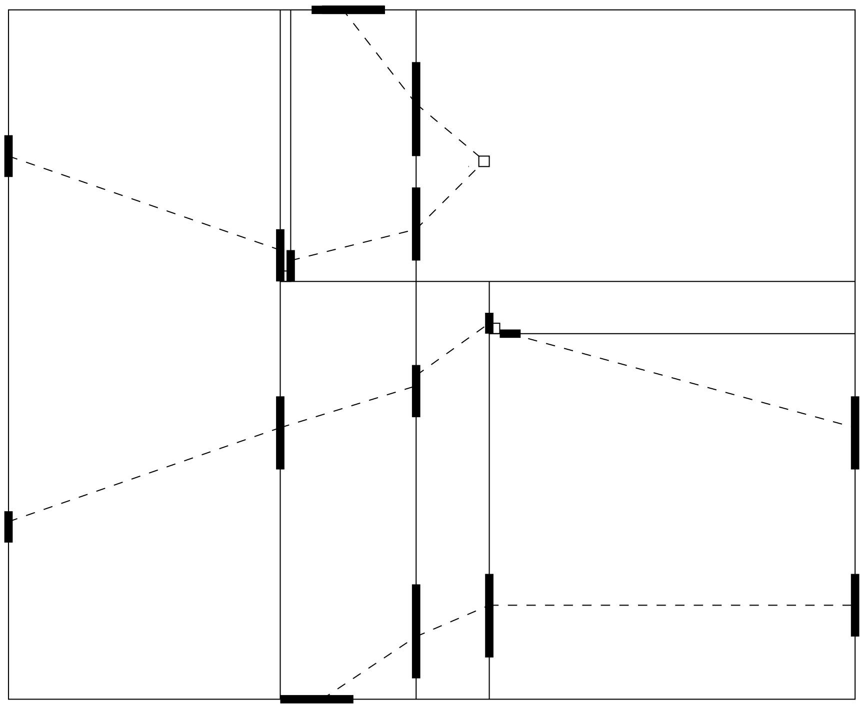


Figure 6: The results of the curve topology algorithm.

rounding it. We can further subdivide the curve so that the non-adjacent subsections 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 do not know of any library which allows the unified exact representations for points and curves that MAPC does. We mention a few related libraries, however, and how they differ from ours.

- *LiDIA* offers (among other things) routines for manipulating exact multi-precision integer and rational numbers, as well as polynomials. We have used LiDIA's rational number routines as the basis of our implementation, but implement our own polynomial classes to allow more specialized (and more efficient) manipulations geared to the specific problems we deal with. LiDIA is not designed for (and does not offer) any geometric data structures of its own.

- *LEDA* offers an extensive and very complete set of routines for dealing with geometric problems. It includes routines such as those in LiDIA for dealing with rational numbers, as well as routines for representing 2D points with floating point or rational coordinates, and performing numerous queries on them. It also includes a way of representing real algebraic numbers by allowing numbers to be represented as the root of another number. LEDA does not contain any support for manipulating (non-linear) curves in the plane, however. This also means that points with algebraic coordinates which are defined as the intersection of such curves cannot be stored or manipulated.

- *CGAL*, like LEDA, offers a framework for performing a number of geometric queries and appears to make use of several LEDA 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* 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 actual geometric data structures, such as for points or curves. It would be interesting to implement the MAPC classes on top of the routines provided in APU, and it is certainly possible that such an implementation could be faster than our current one.

- *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 applications. 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 batteries of 100 random matrices. All of the matrices are 31×31 submatrices of the Sylvester resultant of a polynomial with its first derivative. 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 the polynomial are random numbers.

The first five batteries generate the polynomial's coefficients so that the entries of the matrix are at most b bits, where $b = 16, 32, 53, 64, 128$. (Some entries are a few bits shorter.) These matrices have a strong tendency to be well-conditioned, and we cannot expect this to hold in general. Our second random-number generator tends to produce ill-conditioned matrices that exercise our filter. In the second scheme, a constant $0 < \lambda < 1$ is chosen. Each polynomial coefficient is

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
LiDIA	13.3	44.4	110.1	24.3	30.4	38.0
Inria	1.4	3.7	—	—	—	—
Filter	1.4	1.9	2.1	10.3	26.0	41.0

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

initialized to a 16-bit number. A random number r is chosen uniformly from $[0, 1]$. If $r \leq \lambda$, another 16 random bits are concatenated and the process repeats; otherwise, the process stops. 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 batteries for $\lambda = .1, .2, .3$; entries were no larger than 65, 82, and 116 bits, respectively.

Comparison of methods. In table 7, we show that the four-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`. 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 `bigint`s at all. Our results suggest that if we could remove *all* memory allocation problems (calls to `malloc()`, unnecessary memory cache misses, and so forth) from the determinant routine, we could attain a speedup of as much as 12. (Some techniques for improving the behavior of `bigint`s with respect to memory management are explored in [FV93].)

Utility of filter stages. Table 8 shows the four 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.

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
Zero test	0	0	0	0	0	0
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.

	$b = 16$	$b = 53$	$b = 128$	$\lambda = .1$	$\lambda = .2$	$\lambda = .3$
Zero test	67%	62%	62%	6%	7%	6%
SVD	33%	38%	38%	3%	4%	4%
Inria	—	—	—	9%	10%	12%
LiDIA	—	—	—	82%	79%	78%

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

Time spent in filter stages. We show the percentage of time spent in each of the four filter stages in table 9.

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 are guaranteed to 0.001 accuracy.

Notice that as the cases become more complex, the time spent finding the initial roots 1D roots (but *not* the 1D "box hit" roots) begins to dominate the overall computation time. This is primarily due to high degree and large coefficients. If the

	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 Sturm	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 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 Sturm algorithm and our new algorithm. The time spent in the three main portions of the new routine is given in percentages.

initial polynomials have degrees m and n, and the coefficients have bit lentgths 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 12 presents some example timing results from our curve topology algorithm. Figure 11 shows the four example curves. For each example, figure 12 list the degree of the equation, the number of bits in the coefficient 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 algorithm itself. The drawings show the curves, along with the points found on the curve (including turning points) during the topology algorithm. The points on the curve are connected in order with straight lines. No removal of overlapping bounding boxes is performed.

Notice that the algorithm runs quite fast, and is almost negligible in comparison to the time for computing turning points. The time taken in the algorithm itself seems roughly proportional to the number of turning points, as one might expect. Obtaining the turning points, then, is the primary bottleneck.

6.4 Sorting points along a curve

One application we have applied MAPC to is that of sorting points along a curve. This problem is a key step in certain medial axis computations [CKM98]. Given the representation provided in MAPC the sorting procedure is not very complicated. 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 14 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.

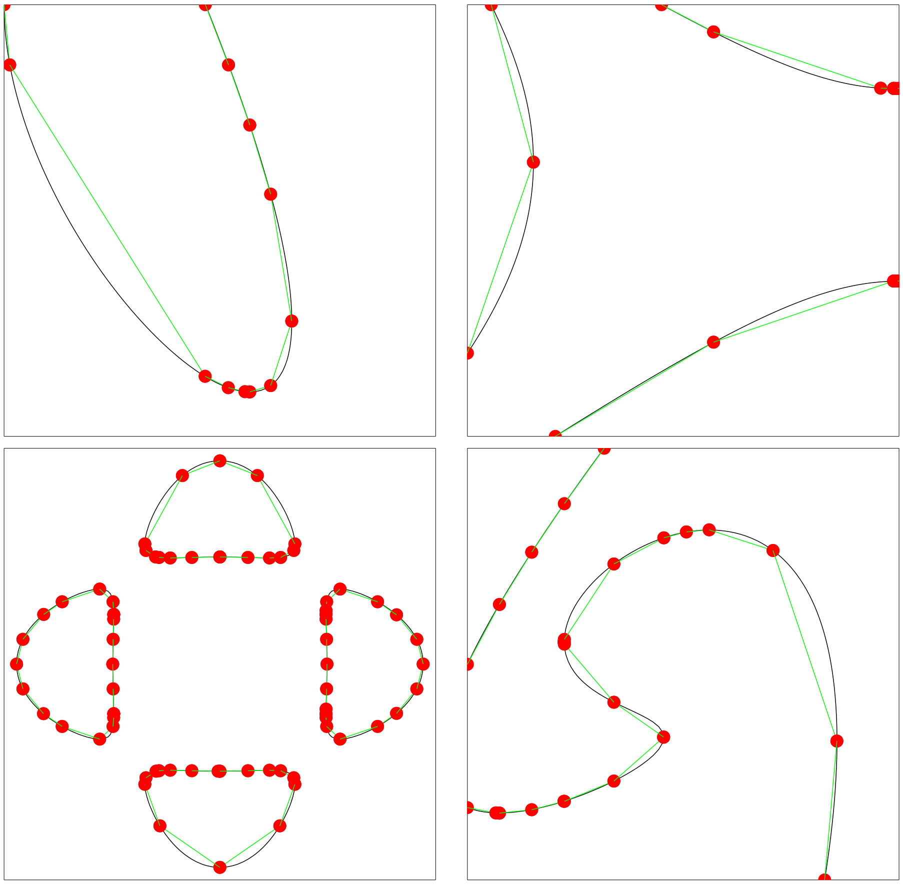


Figure 11: Results of Curve Topology Algorithm. Figures show the four test cases (left to right, top to bottom), along with the points which the topology algorithm returns, connected by lines.

	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 12: Timing Results for Curve Topology Algorithm. The table gives information about each curve within this domain, the time required using our algorithm to isolate the turning points, and the time required to run the topology algorithm.

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: Timing Results for Algebraic Curve Arrangement Algorithm. The table presents the bits required to express the coefficients of the curves (bits in numerator, bits in denominator), the number of faces generated by the arrangement, and the total time taken to compute the arrangement. The curves have maximum degree 4. Drawings of the arrangements can be seen in figure 15.

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. Figure 15 illustrates one such arrangement. The algorithm proceeds in three steps. Initially, all the curves are clipped to the boundary rectangle and their topological type derived from the algorithm specified earlier. All the pairs of curves are then tested for intersection, and the intersection points found. If we treat the set of all boundary, turning, and intersection points as vertices and the curve segments between pairs of these vertices as edges, we can construct an equivalent planar straight line graph. 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 illustrates some timings obtained for various example arrangements. The example shown in figure 15 is case 4 from the table.

7 Conclusion and Future Work

We have presented, here, 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 discussed three new algorithms which have been implemented as a part of MAPC, and have provided some timing results and examples of problems MAPC could be used for.

MAPC is already being used as a component in a solid modeling system, and will hopefully find further application. Future development of the MAPC library itself might 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 3-D 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.

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 adjacency 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.

[Can88] J.F. Canny. *The Complexity of Robot Motion Planning.* ACM Doctoral Dissertation Award. MIT Press, 1988.

[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.

[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.

[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.

[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.

[KP90] D.J. Kriegman and J. Ponce. Computing exact aspect graphs of curved objects: Solids of revolution. *International Journal of Computer Vision*, 5(2):119–135, 1990.

[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.

[She96] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[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.

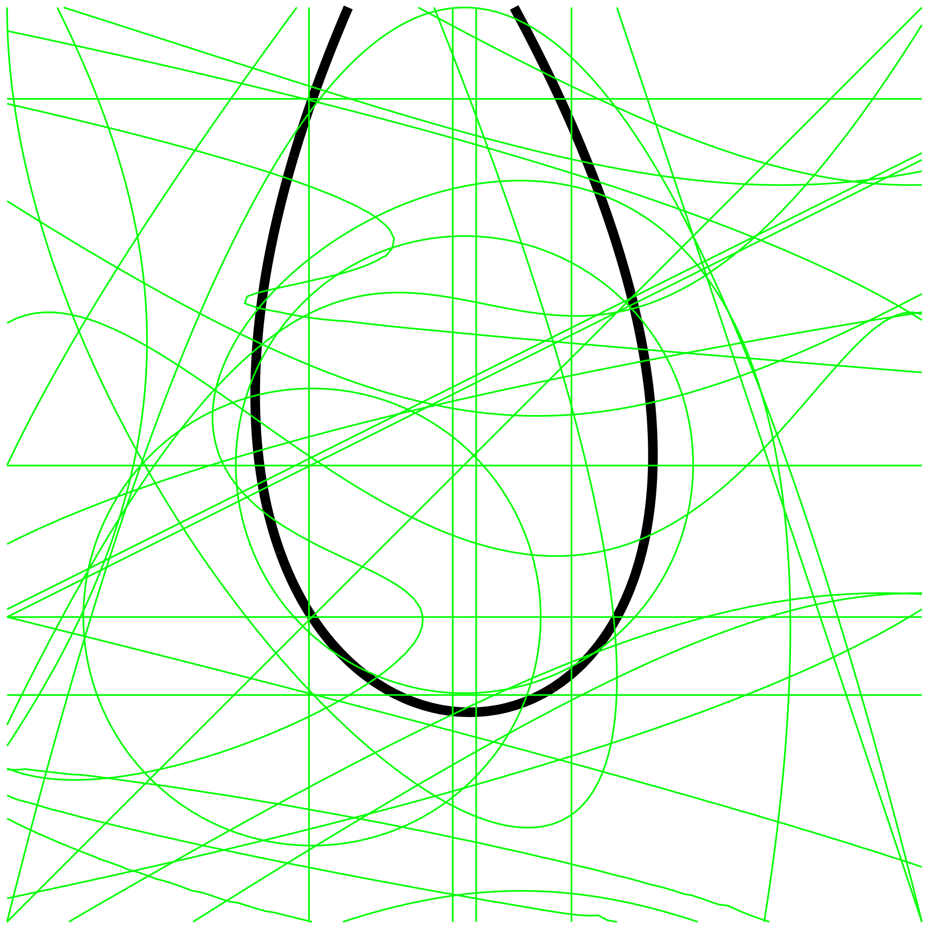


Figure 14: Sorting Points Along a Curve. The curve along which the points are sorted is in bold. The points to be sorted are the 52 intersections of the bold curve with the other 25 curves shown. Finding all intersections takes 102.3 seconds, and the time to perform curve topology on the bold curve and sort the points takes less than one second.

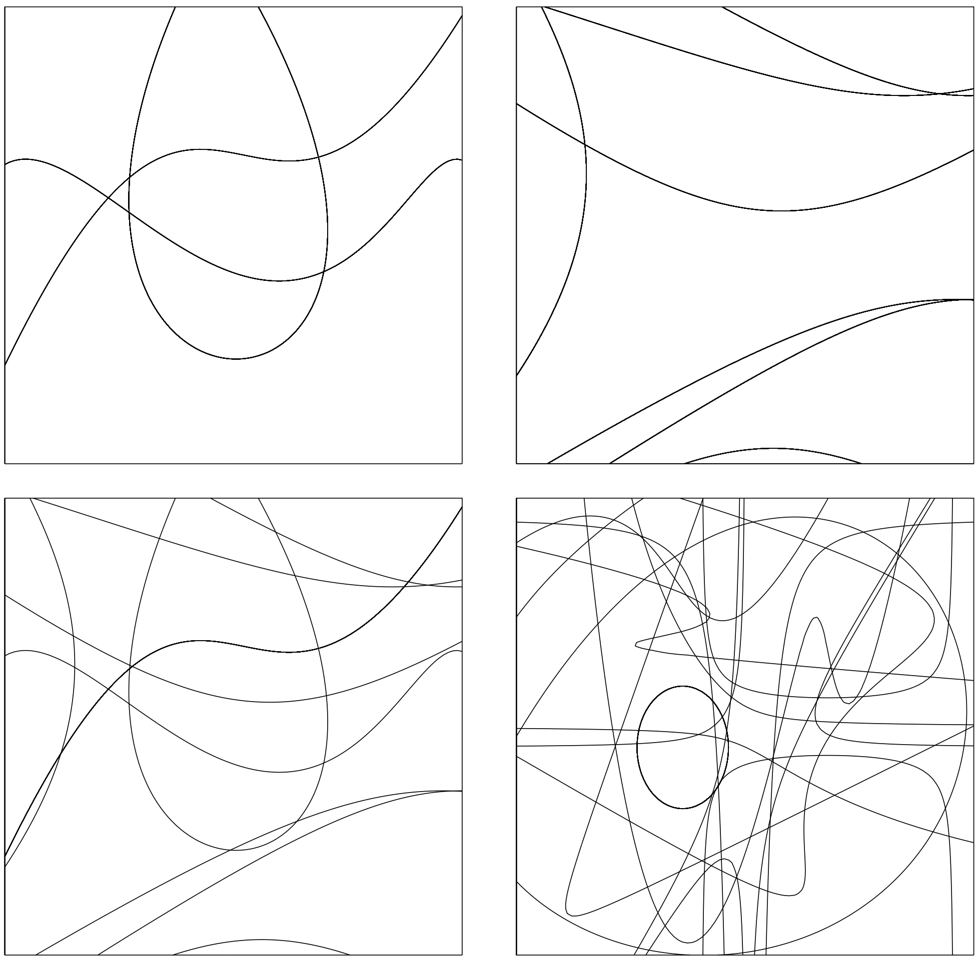


Figure 15: Arrangement of planar algebraic curves. The figure shows the region partitioned into a number of faces by the arrangement of the curves. The application finds all subregions, the portions of curves bounding each subregion, and the connectivity between subregions. The complexity of the curves, the time taken for the four cases (ordered left to right, top to bottom) and the number of faces generated is given in figure 13.

8 Appendix

This section presents some details of the use of the library classes, the box hit configurations for 2D root finding and the region subdivision/connection decision for the curve topology algorithm.

8.1 Use of MAPC Classes

Section 3 gave an overview of the representations used in the classes provided in MAPC. In this section, we will list some of the operations that are available on each of the geometric data structures. Since we are using C++ classes, most of the functionality described is provided as member functions of the classes themselves, although some functions are provided separately.

A number of operations are possible with K_RATPOLYs, some of which are defined only for certain K_RATPOLYs (e.g. only for univariate ones). Some of the key operations are:

- Common arithmetic operations (addition, subtraction, multiplication, division).
- Partial differentiation and sign of a derivative.
- Generation of a Sturm sequence.
- Computing the GCD.
- Conversion to Bernstein basis.
- Substitution of a value or an expression in other variables for one of the variables.
- Interval evaluation (using affine arithmetic).

For K_POINT1Ds and K_POINT2Ds, the following are some of the key operations allowed:

- Obtaining a floating-point approximation to the point.
- Determining an exact interval bounding the point.
- Comparing two points (in either dimension for K_POINT2Ds).
- Sorting the points (on either coordinate for K_POINT2Ds).

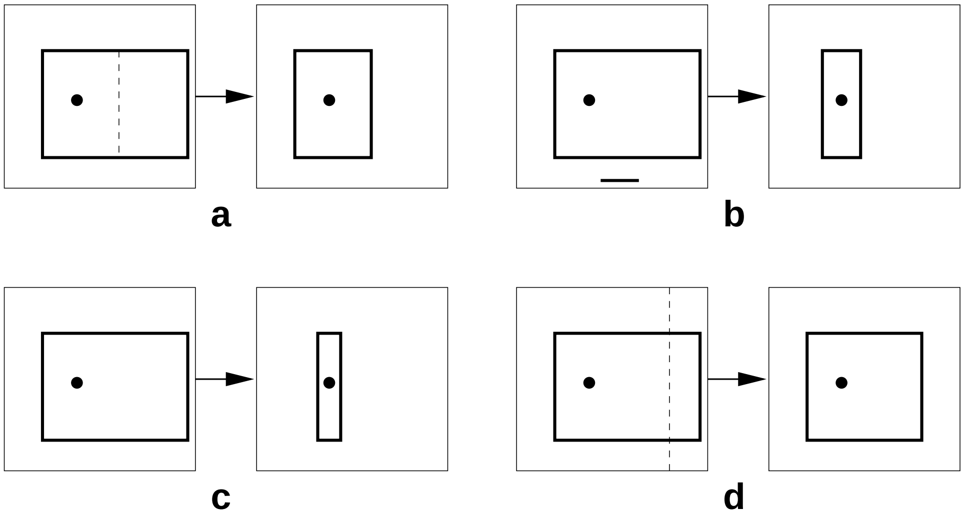


Figure 16: Four operations to reduce the interval surrounding a K_POINT2D. The examples show only reductions in x. The true solution is shown by the dot. a) halving the point. b) contracting the point to a prespecified maximum size (shown by the bar in the first picture). c) shrinking the point to about $\frac{1}{10}$ its original size. d) cutting the point at a specified value.

- Reducing the interval bounding a point. For K_POINT2Ds, this can be done in either (or both) dimension. These are highlighted in Figure 16 Possible operations include:
  - Halving the size of the interval.
  - "Contracting" the interval so that it is smaller than some tolerance,
  - "Shrinking" the interval by some factor.
  - "Cutting" the interval at a particular coordinate value, thus making the interval bounding the point lie either below, above, or on that value.
- Separating two points which are not equal and have overlapping bounding boxes.
- Checking whether two points are equal.

For K_CURVEs, the following are some of the operations which are provided:

- Finding whether some point known to lie on the defining polynomial actually lies on this curve.
- "Splitting" a curve at some point lying on the curve.
- Finding all intersections of the K_CURVE with some other polynomial.
- Finding a bounding box for the curve.
- Sorting points along a curve.
- Determining whether a K_POINT2D lies in a region bounded by a set of K_CURVES.

8.2 Box Hit Configurations

As stated in section 4.2, we need to determine whether or not a box contains a root. Each box will have associated with it some number of "hits" from f and g. These hits can be ordered around the boundary. For each box, we now have one of the following cases. Figure 17 gives a few examples of the types of configurations that may be encountered.

- *No hits from f or no hits from g:* Cases a and b in figure 17 are examples. This means that one of the two curves did not pass through this box. Thus, there can be no intersection between the two curves within this box.
- *Exactly one hit from f (or g):* Case c in figure 17 is an example. This means that that curve tangentially hit the boundary of the box but did not enter it. Thus there can be no intersection inside the box.
- *There are exactly two hits from both f and g:* Let the intersections with f be referred to as A, and those with g as B. If the ordering of the intersections is ABAB (or BABA), as in case d in figure 17, then there must be an intersection within that box. If the ordering is AABB (or ABBA, or BBAA), then two cases are possible: the curves either do not intersect (as in case e of figure 17 or they intersect tangentially (as in case f of figure 17. In some applications, it is quite acceptable to ignore such tangential intersections, in which case AABB cases are never reported as intersections. In other cases, we must handle this event separately, as described below.
- *There are at least two hits from one, and more than two from the other:* Cases g and h from figure 17 are examples. At least one of the curves is

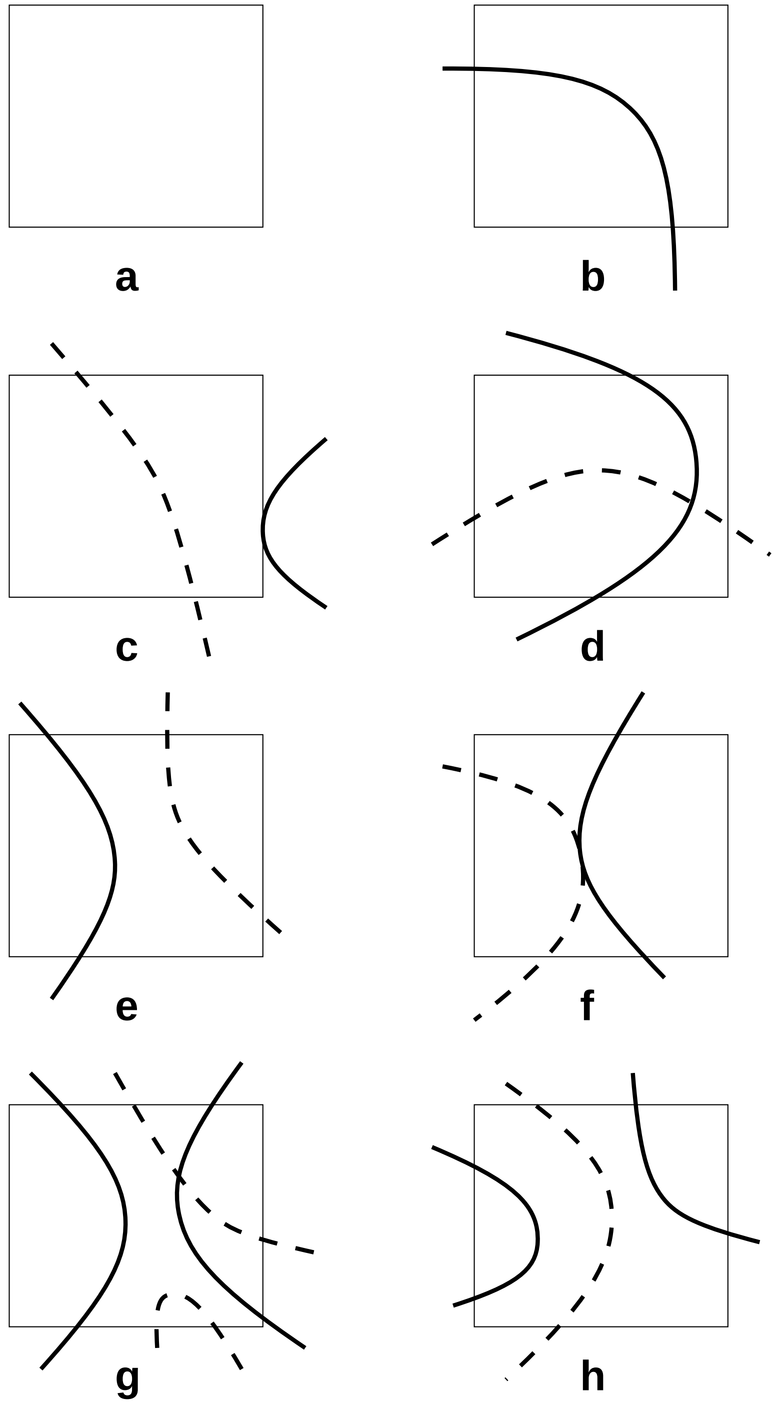


Figure 17: Some potential box hit configurations.

passing through the box multiple times. We can shrink the roots of $X = 0$ and $Y = 0$ corresponding to this box, and again test the four boundaries of the box for hits. At some point, the box will fall into one of the previous categories and we can determine its status conclusively.

In practice, the second and fourth cases are very rare, as is the third case when there is no intersection. Notice also that the tighter the bounds on the roots of $X = 0$ and $Y = 0$, the more likely it is that boxes without intersections inside will fall into case one.

We now discuss how to classify boxes in the third case where there may be a tangential intersection and we wish to know this information. There are two ways this can be done. First of all, the bounds on the corresponding roots of $X = 0$ and $Y = 0$ can be tightened to a level such that we could be guaranteed that if the curves are that close together, they must be intersecting. This would involve the use of gap theorems (see [Can87]), and would likely require a very high precision. A second approach would be to use a multivariate Sturm calculation ([Mil92]), which can exactly count the number of roots within a box. The 2-D Sturm query will return either 0 if there is no root, or 1 if there is one root. In fact, such 2-D Sturm queries could be used in place of the entire algorithm above, as we have previously discussed in [KKM97], but the efficiency of the method described above is significantly superior.

There are a few special cases which we have not discussed. Examples are drawn in figure 18. We outline these here:

- *Intersections with a corner of the box:* It is relatively easy to compute whether $f = 0$ or $g = 0$ intersects the corner of the box. The question then is whether the curve is actually entering the box or just grazing it. As long as the first derivative of the polynomial is not zero (or infinity) at that point, then the answer is simple - just compute the direction the curve is traveling at that point. If the derivative is zero (an extremely rare case), then we treat the box as in the fourth case, and merely compute and test a smaller box. This smaller box will not have the same problem.

- *There are two tangential intersections of the same curve with the boundary of the box, and exactly two intersections of the other curve with the box:* This is an extremely rare case, which it is difficult to imagine happening. Although simply testing a smaller box (as in case 4) would solve the problem, the method we outlined above has no way of distinguishing this case from one where a curve is entering and leaving "normally". A solution would be to test one hit from each curve to make sure that it is not a tangential

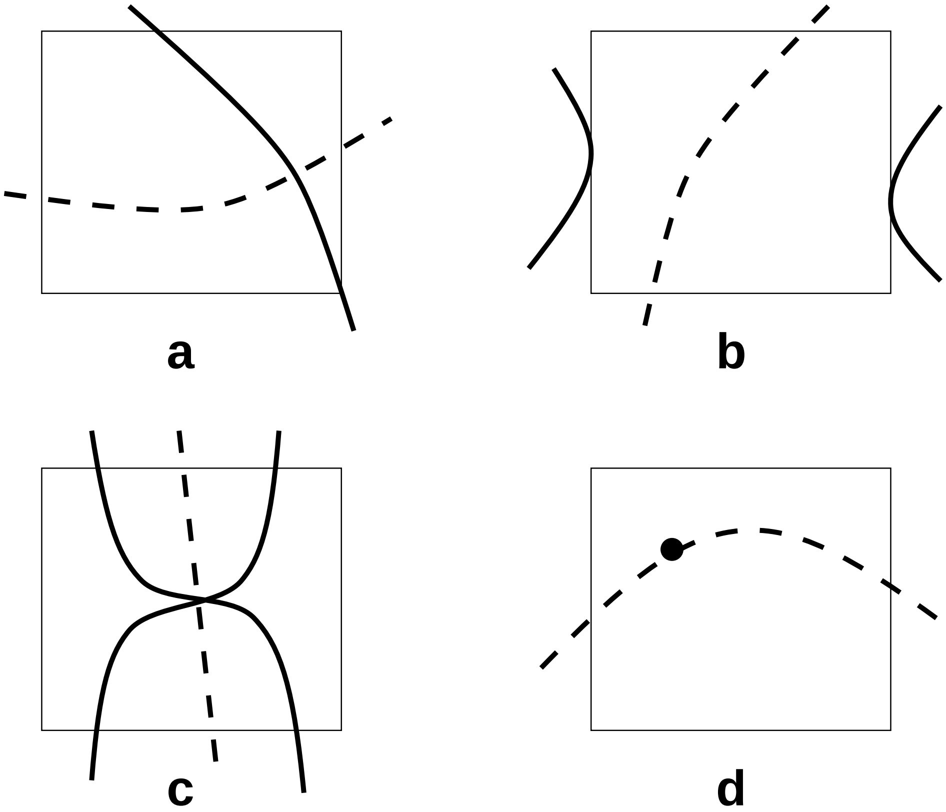


Figure 18: a) An intersection with a corner b) Two tangential intersections c) Intersection at a self-intersection d) Intersection at a point solution

hit whenever it looks like there is an intersection within the box. This involves finding derivatives over an interval, which is not a difficult "fix" to implement, but will decrease efficiency somewhat. We feel that this is such a rare case that it is not worth implementing.

- *One of the curves has a self-intersection, and the other curve intersects it at that point:* If, in the process of shrinking the box in case 4, there remain 4 hits from one curve (say $f = 0$) and 2 from the other in an AABAAB (or similar) pattern, then this could be happening. In such a case, this situation can be decided for certain by using a multivariate Sturm approach (as mentioned above).

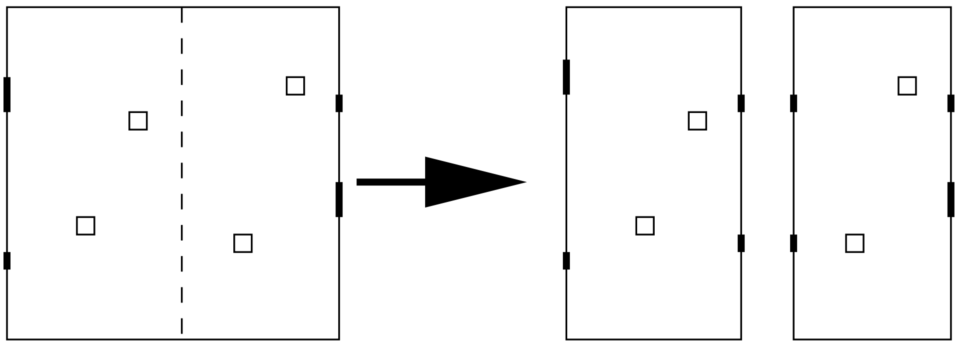
- *One of the curves has only a point solution at the intersection point:* An example of this would be $f(x, y) = x^2 + y^2 = 0$ which has a real solution only at $x, y = (0, 0)$, and $g = 0$ passing through $(0, 0)$. The method described above cannot find such intersections, since the curve $f = 0$ will not intersect a box boundary. If it is important to find such cases in an application, then one can turn to a multivariate Sturm approach (as mentioned above), instead.

In our implementation, we make the assumption that there will not be a self-intersections or point solutions. For a number of applications, though certainly not all, this is a valid assumption.

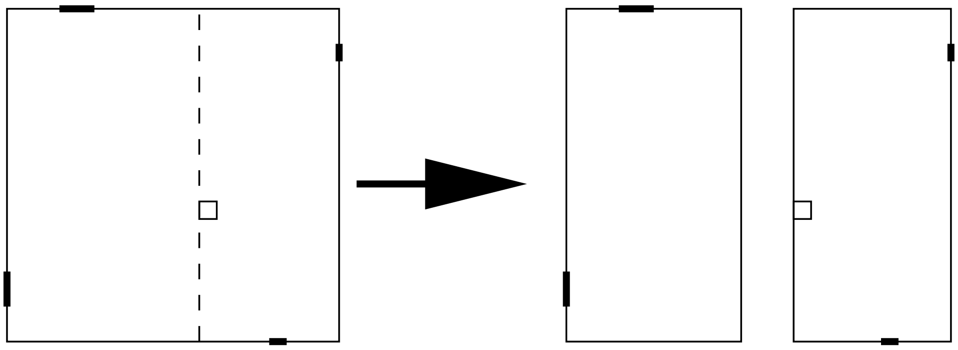
8.3 Curve Topology Configurations

In section 4.3, we described the general procedure for resolving the curve topology. In this section, we describe the details of how the choice to subdivide or connect the edge and turning points is made. We list the different states that the region may be in, and the action that is taken for each. An example figure will be included to illustrate each case.

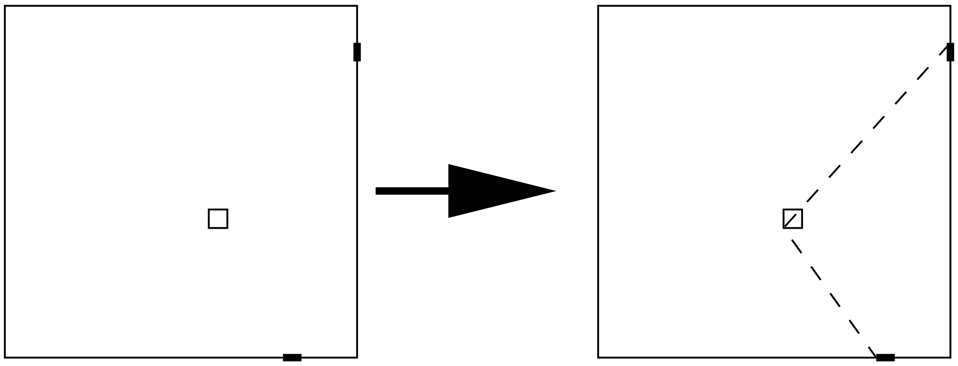
- *The region contains multiple turning points:* Find a vertical or horizontal line that will subdivide the turning points. It is always possible to find such a line, although in the (extremely rare) worst case, the turning points may have to be shrunk first. Thus, a region with n turning points will eventually be subdivided into n regions, each with one turning point.



- *The region contains one turning point and more than two edge points:* The region is subdivided successively along the boundaries of the turning point. If, after all four boundaries have been used for subdivision lines, the region containing the turning point still has more than two edge points, the turning point should be shrunk and the resulting region subdivided again. Eventually, the region must have been subdivided into one region with a turning point and exactly two edge points, and other regions containing only edge points.

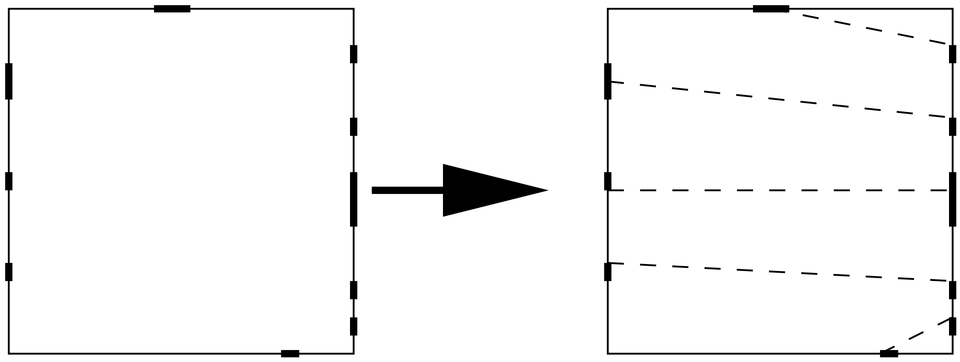


- *The region contains one turning point and exactly two edge points:* First notice that if a region contains a turning point, it must contain at least 2 edge points (since we do not consider point solutions on the curve, and self intersections are either not considered or are treated separately). The curve must pass from one edge point, through the turning point, and leave by the other edge point. Thus, we have the connectivity between all points, and are done with this region.

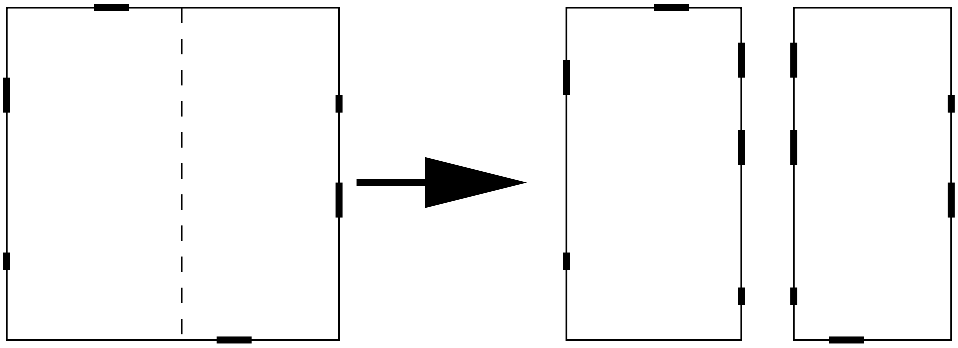


- *The region contains only edge points:* Let L, R, T, and B refer to the *number* of edge points along the left, right, top and bottom edges, respectively. There are two possible cases:

  - *When* $|T - B| = L + R$ *or* $|L - R| = T + B$*:* In such cases (which include any time that there are no edge points along some edge), it is possible to connect the edge points to one another directly. Since there can be no self-intersections in the region, the connections are relatively easy to make.



  - *Otherwise:* The region is subdivided. If $(T + B) < (L + R)$ then subdivide with a vertical line which subdivides the T and/or B points, thus ensuring that $(T + B)$ is smaller in the resulting subregion than in the current one. Otherwise, subdivide the L and/or R points with a horizontal line. Eventually, $T + B$ (or $L + R$) will be at most one, which will fall into the previous case. An exception would occur only for a truly vertical (or horizontal) curve, where $T + B$ would be 2, but in this case, $L + R$ would have to be 0, again yielding the previous case.



We have not discussed how to handle edge points which lie at the corners of the box. It is again (as with root isolation) necessary to make sure that the curve is actually entering and not just grazing the box at the corner, but this can be done by examining the derivative. Notice that since all turning points are isolated, we will never have a tangential intersection along the border of a region. Edge points at the corners can be treated as belonging to either edge (but not both).

As mentioned previously, we will now discuss how to handle self-intersections. Within the algorithm itself, self intersections can be treated as a separate type of turning point. Instead of terminating when there are 2 edge points, termination will occur when there are 4 edge points. The other subdivisions work the same. The difficulty arises in that it is not clear how to represent the connectivity between the four segments that intersect. MAPC represents curves by a series of segments that follow one after the other. There is no procedure for handling the "branching" that must occur to represent self intersections. If this algorithm is implemented in a way that handles self-intersections, the representation of the resulting curves also must have a way to handle self intersections.