Appeared in Proceedings of ISSAC'95

Numeric-Symbolic Algorithms for Evaluating One-Dimensional Algebraic Sets *

Shankar Krishnan Dinesh Manocha

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
USA
{krishnas,manocha}@cs.unc.edu

Abstract: We present efficient algorithms based on a combination of numeric and symbolic techniques for evaluating one-dimensional algebraic sets in a subset of the real domain. Given a description of a one-dimensional algebraic set, we compute its projection using resultants. We represent the resulting plane curve as a singular set of a matrix polynomial as opposed to roots of a bivariate polynomial. Given the matrix formulation, we make use of algorithms from numerical linear algebra to compute start points on all the components, partition the domain such that each resulting region contains only one component and evaluate it accurately using marching methods. We also present techniques to handle singularities for well-conditioned inputs. The resulting algorithm is iterative and its complexity is output sensitive. It has been implemented in floating-point arithmetic and we highlight its performance in the context of computing intersection of high-degree algebraic surfaces.

1 Introduction

The problem of evaluating one-dimensional algebraic sets is fundamental in numeric and symbolic computing. It can be simply stated as finding roots of n affine algebraic equations in $(n+1)$ unknowns. Algebraic sets are widely used for representing objects and constraints in computer graphics, geometric modeling, robotics, computer vision and molecular modeling. Many of the fundamental problems like surface-surface intersection, offsets of curves and surfaces, Voronoi sets generated by curves and surfaces in geometric modeling [Hof89], kinematic analysis of a redundant robot [Cra89], robot motion planning [Can88], object recognition in computer vision [PK92] and conformation space of molecular chains [CH] correspond to evaluating one-dimensional algebraic sets. In most cases we are interested in evaluating all the components in the subset of the real domain. All these problems have been extensively studied in the literature.

It is well known in algebraic geometry that curves of *genus* zero have a rational parametrization [Abh90]. However very few curves have genus zero and in most applications the problem of evaluating algebraic sets corresponds to classifying all components and singularities of the curve and computing a numerical approximation using piecewise straight lines or splines with guaranteed numerical error bounds.

*Supported in part by a Sloan Foundation Fellowship, University Research Award, NSF Grant CCR-9319957, ONR Contract N00014-94-1-0738, and ARPA Contract DABT63-93-C-0048

Main Result: We present numeric algorithms for evaluation of one-dimensional algebraic sets in a subset of the real space. They isolate each component of the curve and evaluate them with guaranteed error bounds. Given a curve, we compute a birationally equivalent algebraic plane curve using resultants. The latter is represented as the singular set of a matrix polynomial (or a ratio of matrix polynomials). Given the matrix representation, we make use of algorithms based on eigenvalues and complex tracing to compute a start point on each component of the curve in the given domain. Furthermore we partition the domain such that each resulting region has a unique component. We present algorithms for evaluating each component, preventing component jumping and handling singularities. The overall algorithm has been implemented in finite precision arithmetic and works well for well-conditioned problems. Its main advantages are *efficiency* and *accuracy*. The resulting algorithm is iterative and its performance is a function of the degree of the algebraic curve, the number of components in the given domain and the accuracy desired. In practice, we have been able to evaluate curves of degree as high as 324 in a few seconds on a SGI Onyx.

Prior Work: There is a considerable amount of work in classic and modern literature related to evaluation of algebraic curves. Every algebraic space curve is birationally equivalent to an algebraic plane curve and the latter can be computed using Gröbner bases [Buc89] and resultants. Given an algebraic plane curve, techniques for desingularization based on quadratic transformations are given in [Wal50, Abh90, AB88]. However, the resulting algorithm can be exponential in the degree of the curve. Algorithms based on Collins' *cylindrical algebraic decomposition* (CAD), [Col75], have been used for evaluating all components of algebraic curves [Arn83, SS83]. However, its worst case complexity is doubly exponential in the number of variables. For plane curves, improved polynomial time algorithms based on CAD have been presented in [AF88, AM88]. However, the exponent in terms of N (the degree of the curve) is rather high. Furthermore, these algorithms are implemented using exact arithmetic, which makes them slow in practice. Other algorithms include those based on Whitney's stratified sets and gap theorems [Can88]. In practice all these algorithms are efficient for low degree curves only. Numerical and finite precision algorithms based on interval arithmetic [Moo79] and homotopy methods [GZ79, Mor92] have been used for evaluating algebraic sets. While the former are slow in practice, the latter have been restricted to zero-dimensional algebraic sets and suffer from problems like component jumping. There is a considerable amount of emphasis in the modeling literature to evaluate surface intersections and offset curves [Hof89, Hof90, SN91, MC91, Hoh91] and in vision literature to compute aspect graphs [PK92]. This includes algorithms for computing all components including the closed loops. However, these algorithms are somewhat restrictive and cannot be used for evaluating general algebraic curves. The combination of resultant formulations and matrix computations have been used for evaluating zero-dimensional algebraic sets in [Laz83, AS86, Man92, Man94]. In this paper, we use them for evaluating one-dimensional algebraic sets.

Organization : The rest of the paper is organized in the following manner. In section 2, we review some literature on resultants and matrix computations and formulate the problem in terms of matrix polynomials. Section 3 describes algorithms for computing a start point on each component of the algebraic curve and in section 4 we highlight the tracing algorithm taking care of component jumping and singularities. We present the implementation and performance of the algorithm for surface intersections in section 5. Section 6 addresses issues of robustness and some measures we take to recover from ill-conditioned inputs. We conclude in section 7.

2 Background

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

$$\begin{aligned} F_1(u,v,w_1,w_2,\ldots,w_{n-1}) &= 0 \\ F_2(u,v,w_1,w_2,\ldots,w_{n-1}) &= 0 \\ &\vdots \\ F_n(u,v,w_1,\ w_2,\ldots,w_{n-1}) &= 0. \end{aligned}$$

We assume that this algebraic set consists of proper components only. Moreover, we are only interested in evaluating the all the components of the curve inside the region $D = [U_1, U_2] \times [V_1, V_2] \times [W_{(1,1)}, W_{(1,2)}] \times [W_{(2,1)}, W_{(2,2)}] \times \ldots \times [W_{(n-1,1)}, W_{(n-1,2)}] \in \Re^{n+1}$. Formally, the functions F_i, $i = 1,2,\ldots,n$, are the components of a vector function $F : D \rightarrow \Re^n$, $D \subset \Re^{n+1}$. The solution to the problem are elements of D that map to the *zero* vector under F.

Since our algorithm works in double precision floating point arithmetic, the results of our algorithm will not always match the output specification given above. Therefore, we modify it to fit our framework. Along with the set of equations, our algorithm requires the definition of a *vector norm* $\| . \|$ on $\Re^n$ and a small positive constant δ ($\geq$ machine precision) as additional inputs. Then the output set consists of all those elements $x \in D$ such that $\| F(x) \| < \delta$. In a number of modeling and visualization applications, values of δ can be provided. It is not clear if it is so in the general case.

We eliminate $n-1$ variables from these equations using multipolynomial resultant algorithms. Almost all the projections are one-to-one and result in a birationally equivalent curve. In our case, we perform a generic linear transformation and eliminate $w_1, \ldots, w_{n-1}$ from the resulting set. The resultant can be expressed in terms of matrices and determinants. In particular, single determinant formulations are known for values of $n = 2,3,4,5,6$ [Dix08, Jou91, MC27, SZ94]. We use $\mathbf{M}(u,v)$ to represent the resulting matrix polynomial. The most general formulation of the resultant expresses it as a ratio of two determinants [Mac02]. Let us denote the top and bottom matrices as $\mathbf{P}(u,v)$ and $\mathbf{Q}(u,v)$ respectively. A similar formulation for $\mathbf{P}(u,v)$ for sparse polynomial systems has been highlighted in [CE93]. In our case, we will make use of the matrix formulation and represent it as an *unevaluated determinant*. It is possible that $\mathbf{P}(u,v)$ and $\mathbf{Q}(u,v)$ are singular, while the resultant is nonzero. In such cases we use the leading non-vanishing minor of $\mathbf{P}(u,v)$ and represent it as $\mathbf{M}(u,v)$. It contains the resultant and an extraneous factor. The algorithm evaluates the resulting algebraic set and substitutes the values back into the original equations to discard the solutions corresponding to the extraneous factor. The degree of the algebraic curve, N, is given by the Bezout or Bernstein bound of the given system of equations.

The plane curve birationally equivalent to the algebraic curve corresponds to the singular set of $\mathbf{M}(u,v)$. For the general Macaulay's formulation the plane curve corresponds to the difference of singular set of $\mathbf{P}(u,v)$ and singular set of $\mathbf{Q}(u,v)$ taking into account the multiplicities of individual factors. For the rest of the paper, we perform a number of numerical computations like determinants, eigenvalues, singular values of $\mathbf{M}(u_i,v_i)$ and similar analysis is applicable in the general case based on the ratio of $\mathbf{P}(u,v)$ and $\mathbf{Q}(u,v)$. Given a point on the plane curve, (u_0, v_0), the corresponding point on the space curve, $(w_{1_0}, w_{2_0}, \ldots, w_{n-1_0})$, is computed using the *kernel* of $\mathbf{M}(u_0,v_0)$ [Man92].

The algorithm for evaluation of algebraic curves computes a start point on each component and traces the resulting component. The *tracing* algorithm uses the local geometry of the curve (like derivative information) to determine successive points. We shall now describe our method to evaluate partial derivatives based on the matrix representation.

2.1 Derivative Computation

We denote the determinant of the matrix $\mathbf{M}(u,v)$ as $D(u,v)$. $D^u(u,v)$ and $D^v(u,v)$ represent the first order partial derivatives with respect to u and v. To be able to trace through the curve we need to evaluate $D(u_1,v_1)$, $D^u(u_1,v_1)$ and $D^v(u_1,v_1)$ for a given point (u_1,v_1) accurately and efficiently. To compute the first and higher order partials, we use a simple variation of Gaussian elimination [MC91]. The basic idea is to compute the partial derivative of each matrix entry at the beginning of computation and update the derivative information along with each step of Gaussian elimination. In this case, we modify the matrix structure such that entry consists of a tuple

$$\mathbf{G}_{ij}(u_1,v_1) = (g_{ij}(u_1,v_1), g^u_{ij}(u_1,v_1), g^v_{ij}(u_1,v_1)),$$

where $g^u_{ij}(u_1,v_1)$ and $g^v_{ij}(u_1,v_1)$ represent the partial derivatives of $g_{ij}(u,v)$ with respect to u and v at (u_1,v_1). The resulting matrix structure is then of the form

$$\overline{M}(u_1,v_1) = \begin{bmatrix} \mathbf{G}_{11}(u_1,v_1) & \ldots & \mathbf{G}_{1n}(u_1,v_1) \\ \vdots & \ldots & \vdots \\ \mathbf{G}_{n1}(u_1,v_1) & \ldots & \mathbf{G}_{nn}(u_1,v_1) \end{bmatrix}.$$

To compute $D(u_1,v_1)$, $D^u(u_1,v_1)$ and $D^v(u_1,v_1)$, we perform Gaussian elimination. We consider the matrix formed by first entry of each tuple (equivalent to $M(u_1,v_1)$) and proceed to compute its determinant using Gaussian elimination. As a side effect we change the entry in the other tuples. Assume we are operating on the ith and kth rows of the matrix. A typical step of Gaussian elimination is of the form $g_{kj} = g_{kj} - \frac{g_{ki}}{g_{ii}} g_{ij}$, where g_{kj} represents the element in the k^{th} row and j^{th} column of the matrix. In the new formulation this step is replaced by three steps: $g_{kj} = g_{kj} - \frac{g_{ki}}{g_{ii}} g_{ij}$, $g^u_{kj} = g^u_{kj} - \frac{(g^u_{ki} g_{ij} + g_{ki} g^u_{ij}) g_{ii} - (g_{ki} g_{ij}) g^u_{ii}}{(g_{ii})^2}$, and $g^v_{kj} = g^v_{kj} - \frac{(g^v_{ki} g_{ij} + g_{ki} g^v_{ij}) g_{ii} - (g_{ki} g_{ij}) g^v_{ii}}{(g_{ii})^2}$. We make a choice for the pivot element based on the first tuple (i.e. g_{ij} entry). After Gaussian elimination is complete, we compute $D(u_1, v_1), D^u(u_1, v_1)$ and $D^v(u_1, v_1)$ as $D(u_1, v_1) = \prod_{i=1}^n g_{ii}$, $D^u(u_1, v_1) = D(u_1, v_1) \sum_{i=1}^n \frac{g^u_{ii}}{g_{ii}}$, and $D^v(u_1, v_1) = D(u_1, v_1) \sum_{i=1}^n \frac{g^v_{ii}}{g_{ii}}$. This procedure can be easily extended to compute the higher order partial derivatives as well. The accuracy of the resulting algorithm is improved by partial and complete pivoting as well. For the general Macaulay's formulation, we individually compute the partials of the determinants of $\mathbf{P}(u, v)$ and $\mathbf{Q}(u, v)$ and make use of quotient rule to compute the partials of the resultant.

3 Computation of Start Points and Loop Characterization

In this section, we shall describe algorithm for computing the start points on every component of the curve. This is very critical for the tracing algorithm. Inside the domain $[U_1, U_2] \times [V_1, V_2]$, the plane curve consists of two types of components: *open* and *closed*. *Open* components have at least one point lying on the boundary of the domain. *Closed* components, on the other hand, lie completely within the rectangular domain. We refer to them as *loops*. In practice, finding start points on loops is significantly harder than those on open components.

3.1 Start Points on Open Components

From the definition of open components, it is clear that its endpoints must lie on the boundary of the domain. Starting points on all these components can be obtained by solving the system after substituting one of $u = U_1$, $u = U_2$, $v = V_1$ and $v = V_2$ into $\mathbf{M}(u, v)$. This results in a univariate matrix $\hat{\mathbf{M}}(u)$ or $\hat{\mathbf{M}}(v)$ depending on the variable substitution. Without loss of generality, we can assume that the variable v was substituted. The resulting matrix $\hat{\mathbf{M}}(u)$ can then be written as $\hat{\mathbf{M}}(u) = u^d \hat{M}_d + u^{d-1} \hat{M}_{d-1} + \ldots + u\hat{M}_1 + \hat{M}_0$, where the $\hat{M}_i$'s are numeric matrices. We need to find all the solutions of this matrix polynomial. The roots of $\hat{\mathbf{M}}(u)$ have a one-to-one correspondence with the eigenvalues of the companion matrix $\mathbf{C}$ [Man92].

$$\mathbf{C} = \begin{bmatrix} 0 & I_n & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_n \\ -\overline{M}_0 & -\overline{M}_1 & -\overline{M}_2 & \ldots & -\overline{M}_{d-1} \end{bmatrix} \qquad (1)$$

where $\overline{M}_i = \hat{M}_d^{-1} \hat{M}_i$ [GLR82].In case $\hat{M}_d$ is singular or ill-conditioned, the problem is reduced to a generalized eigenvalue problem [Man92]. Algorithms to compute all the eigenvalues are based on QR orthogonal transformations [GL89]. They compute all the real as well as complex eigenvalues. To compute a real or complex subset of the roots, iterative algorithms are given in [Man94]. If there are few real solutions (two or three in the domain), the latter algorithm is significantly faster than the QR algorithm. For general Macaulay's formulation, eigenvalues computations are performed on $\mathbf{P}(u, v_i)$ and $\mathbf{Q}(u, v_i)$ and we take a difference of the two sets accounting for the multiplicities of the individual roots.

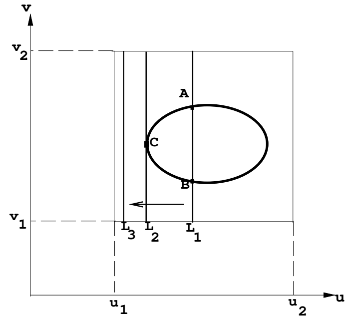

Figure 1: Characterization of loops

We label this method as *boundary-value computation* and it gives both the endpoints of every open component. The difficulty in identifying start points on closed components lies in the fact that loops have no such simple characterization as the one for open components. However, we show that we can use a simple algebraic property that would guide us to at least one point on every loop.

3.2 Identifying Start Points on Loops

The curve $\mathbf{D}(u, v)$ is an algebraic plane curve in the complex projective plane defined by u and v. We are, however, interested only in finding the part that lies in the portion of the real plane defined by $(u, v) \in [U_1, U_2] \times [V_1, V_2]$. If we relax this restriction so that one of the variables, say v, can take complex values, the intersection curve is defined as a continuous set consisting of real and complex components. Before we give our loop characterization, we introduce some basic definitions.

Definition 1 **Turning points** *are points on the plane curve where the tangent vector, as projected in the (u, v) space, is parallel to the u or v parameter axis. Moreover, one of the partial derivatives (with respect to u or v) of the curve is 0. For eg.,* **u**-*turning points are points where the tangent is parallel to the v axis. We classify u-turning points into* left u-turning points *and* right u-turning points. *A point (u^1, v^1) is a* left u-turning point *if the curve goes into the complex domain in the left neighborhood of u_1 ($u = u_1 - \delta$, where δ is a small positive value). A point (u_1, v_1) is a* right u-turning point *if the curve goes into the complex domain in the right neighborhood of u_1 ($u = u_1 + \delta$).*

Lemma 1 *If the curve in the real domain $[U_1, U_2] \times [V_1, V_2]$ consists of a closed component, then two arbitrary complex conjugate paths meet at one of the real points (corresponding to a turning point) on the loop.*

Proof: The proof is obvious from the continuity properties in the complex space. Since the loop is isolated in the real space, it is connected to other components in the complex space. Consider the loop shown in Fig.1. At line L_1, the roots of v occur at points A and B. The roots come closer together as u is changed from L_1 to L_2. At L_2, the roots coincide to form a double root. This is precisely where the turning point occurs. When u is changed from L_2 to L_3, the double root corresponding to v becomes complex. And since all the coefficients of the curve are real, the complex roots must occur in conjugate pairs.

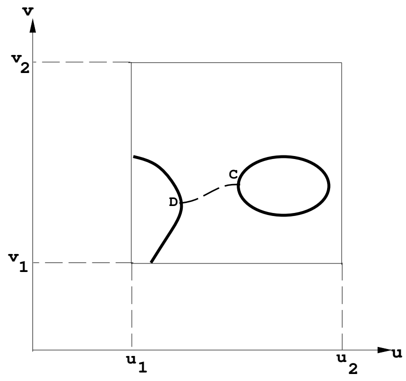

Figure 2: Possible path in complex space

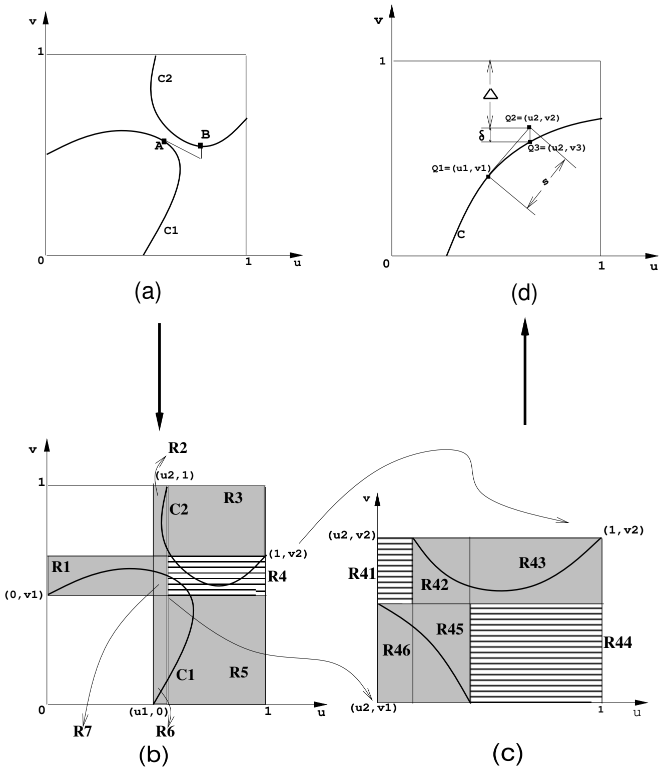

Figure 3: (a)Component jumping (b)First-level decomposition (c)Second-level decomposition (d)Tracing step

□

Since we need only one point on every loop to trace it completely, we restrict ourselves to left u-turning points only. The domain has changed from the real plane to a three dimensional space formed by u, v_r and v_i, where v_r and v_i are the real and imaginary values of v. We use boundary-value computation followed by complex tracing (tracing in complex space) to determine turning points on loops. Since all complex roots occur in conjugate pairs, it suffices to trace only one of them (with positive imaginary values). However it is not enough to only trace the complex paths obtained after performing boundary-value computation on $\mathbf{M}(U_1, v)$. For example, consider Fig.2. A complex path arises from point D (a right-turning point) and ends in C (left-turning point of the loop). The complex path is shown in dotted lines. This path will never be detected by the previous method.

When a complex path touches the real plane the imaginary part (of v) must pass through some small constant value ϵ before reducing to zero. These are precisely the common points of the curve with the plane $v_i = \epsilon$. In other words, we are trying to find all the real solutions to the equation $DET(\mathbf{M}(u, v_r + i\epsilon)) = 0$. Expanding out the expression and collecting the real and imaginary terms we can write

$$DET(\mathbf{M_r}(u, v_r) + i\mathbf{M_i}(u, v_r)) = 0 \qquad (2)$$

It is easy to show that the solutions (u, v_r) satisfying equation (2) also satisfy $DET(\mathbf{R}(u, v_r)) = 0$, where

$$\mathbf{R}(u, v_r) = \left[\begin{array}{cc} \mathbf{M_r}(u, v_r) & -\mathbf{M_i}(u, v_r) \\ \mathbf{M_i}(u, v_r) & \mathbf{M_r}(u, v_r) \end{array} \right] \qquad (3)$$

As before, the solutions to (3) can be posed as the singular set of matrix $\mathbf{R}(u, v_r)$. This singular set is a discrete point set and the order of the matrix is twice that of $\mathbf{M}(u, v)$.

Initially we form the companion matrix of $\mathbf{R}(u, v_r)$, C_r, similar to the one in Eq.(1). We compute all the eigenvalues of C_r at $u = U_1$ (we expect all of them to be complex). We use the ones with positive imaginary points as starting points and trace all the paths in increasing u direction until it either crosses the $u = U_2$ plane or become real. All the real values of v_r are points lying very close to the turning points of the intersection curve. They are denoted by (u_r, v_r). Then (u_r, v_r) is used as an initial guess to converge to the turning point using inverse power iterations. When complex tracing is done, all the turning points which are potentially part of loops are obtained. The details of tracing using inverse power iterations are presented in the next section.

4 Tracing

Given the start points, we evaluate the curve using our tracing algorithm. A number of algorithms for tracing based on local iterative methods have been used in homotopy methods, surface interrogations and solutions of differential equations [Hof89, Mor92]. Given a point on the curve, an approximate value of the next point is obtained by taking a small step size in a direction determined by the local geometry of the curve. Based on the approximate value, these algorithms use local iterative methods like Newton's method to trace back on to the curve. Given a start point, these algorithms are applicable to trace algebraic curves as well. However, the three main issues concerning tracing algorithms are:

1. Converging back on to the curve.
2. Preventing *component jumping*.
3. Ability to handle singularities and trace through multiple branches.

The convergence problems arising from the behavior of Newton's method are well known. It is rather difficult to predict the convergence of Newton's method on high degree equations. *Component jumping* can occur when two components of the curve are relatively close to each other as shown in fig.3(a). In this case, the tracing algorithm can jump from point A on component C1 to point B on component C2. Most implementations circumvent this problem by choosing *very small* and *conservative* step sizes. But this still cannot guarantee correctness and moreover, slows down the algorithm. Singularities are points where the curve self-intersects or the tangent vector vanishes. They typically lead to multiple branches around the singular point. The tracing algorithm has to determine these singular points efficiently and trace all the branches.

We present an efficient tracing algorithm that can resolve all these issues most of the time. In particular we introduce a technique called *component splitting* and tracing based on *inverse power iterations*. Singularities are also handled efficiently for well-conditioned inputs.

4.1 Component Splitting

After performing boundary-value computation and loop detection, a sequence of points are obtained on the curve ($(u,v) \in [U_1,U_2] \times [V_1,V_2]$) which either correspond to starting points on open components or some point on loops. Using these points the plane curve is traced completely without missing any important curve features. The idea behind *component splitting* is that if there are only two boundary points inside a region with no loops, these points belong to the same component of the curve. Further there exists exactly one component of the curve inside this region. Therefore, the purpose of the algorithm is to subdivide the original domain into smaller regions such that each region contains exactly one curve component.

We now describe the working of component splitting. The input into this routine is a rectangular domain, specified as $[L_u,H_u] \times [L_v,H_v]$, and a set of points, S, on the curve inside this domain. S covers all the components of the intersection curve inside the domain. If the cardinality of S is two, then we are assured of a single curve component and the decomposition terminates. If the cardinality of S is greater than two the algorithm subdivides the domain along *isolines* (lines of constant u or v in the domain) determined by the values of points of S. The isolines chosen at every point could either be a *u-isoline* ($u = u_1$) or a *v-isoline* ($v = v_1$). The algorithm arbitrarily chooses the v-isoline to subdivide the domain. If subdivision is not possible (all the points in S have v coordinates as L_v or H_v), then u-isoline is chosen for subdivision. In the process, new points corresponding to the intersections of the isolines with the curve are generated and inserted into the appropriate regions. The component splitting algorithm is then applied recursively to each newly created region.

Subdivisions of domains are not carried out indefinitely. If the dimensions of a domain become smaller than a specified tolerance, the subdivision is stopped and checked for singularities. Informally, singularities are points on the curve where the curve self-intersects or has multiple branches. In the presence of singularities (excluding cusps), no level of decomposition can produce subdomains with one simple curve component unless the singular point is determined accurately. If the algorithm is unable to isolate single curves in a domain after repeated levels of subdivision, then one of two cases can occur.

- The curve has a singularity, or
- Two components of the curve are very close.

At this point, minimization of an energy function $E(u,v)$ distinguishes the two cases.

$$E(u,v) = (D(u,v)^2 + D^u(u,v)^2 + D^v(u,v)^2) \qquad (4)$$

where $D(u,v)$ is the determinant of $\mathbf{M}(u,v)$, and $D^u(u,v)$ and $D^v(u,v)$ are the partial derivatives of $D(u,v)$ with respect to u and v respectively. The minimization is applied with the midpoint of the region as the initial point. A minimum value of zero (with tolerance) corresponds to a singularity. A non-zero minimum value means that the curve has two very close components. If there is a singularity, then subdivision is done at the singular point and component splitting is performed at each subdomain. Singularities of a high degree curve can be very sensitive to small input perturbations and floating point errors. Therefore, the algorithm reports a singularity if the minimum value obtained is smaller than a user-specified value. This method is susceptible to numerical errors especially if the singularities lie very close to each other. We, however, believe that such pathological cases are rare in practice.

The pseudocode for the above algorithm is given below:

- **ComponentSplitting**(*domain, Xsection_points, tolerance*)
  1. If (there are only two *Xsection_points*) trace the curve inside the region and return.
  2. If (region size is smaller than *tolerance*)
     - Apply singularity criterion.
     - If there is a singularity
       * Subdivide the domain at the singular point along both axes.
       * Find all intersection points along the subdivided curves.
       * for each subregion, do ComponentSplitting(subregion, new_points, *tolerance*).
       * Return.
  3. If (domain convergence is slow)
     - Divide the domain at midpoint of one of the intervals.
     - Compute the intersections of the curve with the dividing line.
     - For each of the two subregions, do ComponentSplitting(subregion, new_points, *tolerance*).
     - Return.

     else
     - Divide the domain along isolines from every *Xsection_points*. For the point (L_u, v^1), the corresponding line is $v = v^1$.
     - Compute the intersection of the curve with each such line.
     - For each subregion, do ComponentSplitting(subregion, new_points, *tolerance*).
     - Return.

After performing this algorithm, a set of curves traced inside each region is obtained. Some of these are parts of the same curve component. By matching their endpoints, they are connected appropriately to obtain the original curve in the $[U_1,U_2] \times [V_1,V_2]$ domain. This algorithm guarantees

- No component jumping - tracing is performed only inside a region that is guaranteed to contain just one curve.
- Singularity detection - During all stages of the algorithm, geometrically isolated singular points are always bracketed.

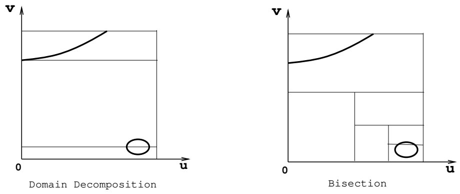

Figure 4: Comparing our algorithm vs. Bisection

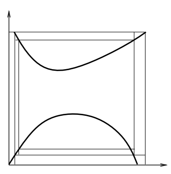

Figure 5: Case of slow convergence of our algorithm

Two levels of the decomposition algorithm have been highlighted on a particular curve in Figs.3(b) and 3(c). The number of levels of subdivision performed by the algorithm depends directly on the presence of singularities or close components in the curve. The decomposition step is similar in nature to that of interval arithmetic based algorithms. However, for most cases our algorithm performs fewer levels of decomposition. Interval arithmetic based algorithms [Sny92] perform a number of decompositions as a function of the accuracy parameter. These algorithms are usually robust, but are slow in practice. Our algorithm performs about an order of magnitude faster than interval arithmetic based methods (as applied to surface intersections).

4.1.1 Performance and Convergence

Fig.4 provides a comparison between ordinary bisection (corresponding to dividing the domain into equal halves at each step) and our component splitting algorithm. It can be seen that in the case depicted by the figure, our method performs much better than bisection. In fact, on an average, our method achieves the desired level of subdivision much faster than bisection. This is because our method is a form of guided subdivision as opposed to blind partitioning adopted by bisection.

However, there are instances when the algorithm does not reduce the region size appreciably. This usually happens when the set of intersections are very close to the corners of the region. One such example is illustrated in fig.5. These cases can be detected easily though. When such instances are encountered, bisection is performed once on the domain to break the symmetry (of points in set S). Component splitting is then performed on each half. The step size of tracing is determined by the size of each region.

The algorithm given above is used to partition the domain of the curve into regions with a single curve component. Its complexity is a function of the number of components and the separation of the components into various regions. For most practical cases, there are a few and well-separated components in the real domain and the algorithm performs well for such cases. In many ways the underlying philosophy is rather similar to CAD based algorithms for partitioning the domain into regions. Our algorithm uses an efficient and accurate zero-dimensional solver [Man94] and works well using finite precision algorithm. On the other hand, the CAD based algorithm [Arn83] compute all the extremal and turning points using purely symbolic methods and exact arithmetic. Even though this method guarantees that the solution is always topologically reliable, they are impractical because of their large memory requirements and poor efficiency.

4.2 Tracing in lower dimension

After component splitting, the entire domain ($[U_1, U_2] \times [V_1, V_2]$) is subdivided into smaller regions each with exactly one curve segment. It also returns the two endpoints of the curve inside the region. Starting from one of the endpoints, the *tracing* algorithm computes successive curve points using the local geometry of the curve until the other endpoint is reached. Let the component be C. Given a point $\mathbf{Q}_1 = (u_1, v_1)$ the skeleton of the tracing algorithm is given below. Furthermore, it is required that any point on the piecewise representation of the curve is not more than ϵ apart from the curve.

- Compute $D^u(u_1, v_1)$ and $D^v(u_1, v_1)$, the partial derivatives of the curve with respect to u and v, respectively. This is the vector (on the plane) normal to the plane curve.
- Given the normal vector, find the unit vector corresponding to the tangent. Let this vector be (t_u, t_v).
- Find an approximate point $\mathbf{Q}_2 = (u_2, v_2)$, where $u_2 = u_1 + t_u * S$, and $v_2 = v_1 + t_v * S$, where S is the step size and $S \leq \epsilon$.
- Using (u_2, v_2), converge back to the curve at $\mathbf{Q}_3 = (u_2, v_3)$, if $\mid t_u \mid > \mid t_v \mid$, or to $\mathbf{Q}_3 = (u_3, v_2)$, if $\mid t_v \mid > \mid t_u \mid$ using *inverse power iterations.*

A single tracing step is shown in fig.3(d). The two main components of the tracing algorithm are the choice of step size and tracing back to the curve component using inverse power iterations. We explain each of them in detail. For the rest of the analysis we will assume that $\mathbf{Q}_3 = (u_2, v_3)$.

Inverse Power Iterations: In a single step of the tracing algorithm, we need to compute the eigenvalue of $\mathbf{M}(u_2, v)$ which is closest to v_2 (fig.3(d)). As a result, we compute the companion matrix $\mathbf{C}$ from $\mathbf{M}(u_2, v)$ (see eq. (1)) and set $s = v_2$. Therefore, we need to compute the smallest eigenvalue of the matrix $\mathbf{C} - s\mathbf{I}$. The smallest eigenvalue of $\mathbf{C} - s\mathbf{I}$ corresponds to the largest eigenvalue of $(\mathbf{C} - s\mathbf{I})^{-1}$. Instead of computing the inverse explicitly (which is numerically unstable), we use inverse power iterations [GL89].

To solve the matrix system efficiently, we use LU decomposition of the matrix $(\mathbf{C} - s\mathbf{I})$ using Gaussian elimination. We also make use of the structure of the matrix to reduce its complexity. Given s, let $\mathbf{B} = \mathbf{C} - s\mathbf{I}$. $\mathbf{B}$ is of the form:

$$\mathbf{B} = \begin{pmatrix} \alpha_1 \mathbf{I}_n & \mathbf{I}_n & \mathbf{0} & \dots & \mathbf{0} \\ \ddots & \dots & & & \\ \mathbf{0} & \mathbf{0} & \dots & \alpha_1 \mathbf{I}_n & \mathbf{I}_n \\ \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 & \dots & \mathbf{P}_m \end{pmatrix}$$

where α_1 is a function of s. The LU decomposition of $\mathbf{B}$ has the form:

$$\mathbf{B} = \begin{pmatrix} \alpha_1 \mathbf{I}_n & 0 & \ldots & 0 \\ 0 & \alpha_1 \mathbf{I}_n & \ldots & 0 \\ \ddots & \ldots & & \\ \mathbf{R}_1 & \mathbf{R}_2 & \ldots & \mathbf{L}_m \end{pmatrix} \begin{pmatrix} \mathbf{I}_n & \frac{1}{\alpha_1}\mathbf{I}_n & \ldots & 0 \\ 0 & \mathbf{I}_n & \ldots & 0 \\ \ddots & \ldots & & \\ 0 & 0 & \ldots & \mathbf{U}_m \end{pmatrix}$$

where $\mathbf{L}_m$ and $\mathbf{U}_m$ correspond to the LU decomposition of $\mathbf{R}_m$. $\mathbf{R}_i$'s can be easily computed from the $\mathbf{P}_i$'s. LU decomposition can sometimes face numerical problems if the matrix B is ill-conditioned. In such cases, QR factorization can be used, where Q and R are orthogonal and upper triangular matrices respectively.

A key property of inverse power iteration is that it converges to the eigenvalue closest to s. If the closest eigenvalue to s is a complex conjugate pair, the power method does not converge to any real value. In such cases, the tracing algorithm chooses a smaller step size for computation.

Step Size Computation: The step size S is chosen to prevent component jumping. To avoid component jumping the following constraints are imposed on $\mathbf{Q}_2$. Let the closest distance of $\mathbf{Q}_2$ to the domain boundary be Δ as shown in fig.3(d). As a result, any point on any other component of the curve is at least Δ away. Furthermore, the distance δ from $\mathbf{Q}_2$ to $\mathbf{C}$ is at most ϵ. If $\delta < \Delta$, the inverse power iteration guarantees that after the convergence of power iteration the resulting point is still on $\mathbf{C}$. Therefore, a bound on the choice of stepsize is given by the condition $\delta < S < \Delta$. We initially choose a value of S and check whether $S < \Delta$. If this constraint is not satisfied we refine the value of S using a binary search over the range $[0, S]$. Thus making use of component splitting and inverse power iterations, we avoid component jumping during tracing.

5 Implementation and Performance

The algorithm has been implemented and its performance was measured on a number of models. The algorithm uses existing EISPACK [GBDM77] and LAPACK [ABB+92] routines for some of the matrix computations. At each stage of the algorithm, we can compute bounds on the accuracy of the results obtained based on the accuracy and convergence of numerical methods adopted like eigenvalue computation, power iteration and Gaussian elimination. The algorithm was implemented on an SGI Onyx workstation. This workstation has 128MB of main memory and a specFP rating of 97.1. In most applications, the number of components in the real domain is small and well separated and the algorithm performs very well in such cases.

5.1 Application to Surface Intersection

In this section, we shall discuss the application of our algorithm to a particular case, surface-surface intersection. In most applications, the problem appears as intersection of two parametric surfaces. A special case of parametric surface is the *tensor-product Bézier* surface (simply called Bézier patch). A Bézier patch, $\mathbf{F}(s,t)$, of degree $m \times n$ is represented as: $\mathbf{F}(s,t) = \Sigma_{i=0}^{m} \Sigma_{j=0}^{n} \mathbf{V}_{ij} B_i^m(s) B_j^n(t)$, where $\mathbf{V}_{ij} = \langle w_{ij}x_{ij}, w_{ij}y_{ij}, w_{ij}z_{ij}, w_{ij} \rangle$ are the *control points* of the patch in homogeneous coordinates and $B_i^m(s) = \begin{pmatrix} m \\ i \end{pmatrix} s^i(1-s)^{m-i}$ is the Bernstein polynomial. The domain of the surface is defined on the unit square $0 \leq s,t \leq 1$ in the (s,t) plane.

Given two Bézier surfaces, $\mathbf{F}(s,t)$ and $\mathbf{G}(u,v)$

$$\begin{aligned} \mathbf{F}(s,t) &= (X(s,t), Y(s,t), Z(s,t), W(s,t)) \\ \mathbf{G}(u,v) &= (\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v)) \end{aligned}$$

represented in homogeneous coordinates, their intersection curve is defined as the set of common points in 3-space and is given by the vector equation $\mathbf{F}(s,t) = \mathbf{G}(u,v)$. This results in the following set of three equations in four unknowns:

$$\begin{aligned} F_1(s,t,u,v) = X(s,t)\overline{W}(u,v) - \overline{X}(u,v)W(s,t) &= 0 \\ F_2(s,t,u,v) = Y(s,t)\overline{W}(u,v) - \overline{Y}(u,v)W(s,t) &= 0 \quad (5) \\ F_3(s,t,u,v) = Z(s,t)\overline{W}(u,v) - \overline{Z}(u,v)W(s,t) &= 0, \end{aligned}$$

and the domain of the intersection curve is $(s,t,u,v) \in [0,1] \times [0,1] \times [0,1] \times [0,1]$. The variables s and t are eliminated using Dixon's resultant [Dix08]. If the patch $\mathbf{F}(s,t)$ is of degree $m \times n$, then the Dixon's resultant obtained by eliminating s and t from F_1, F_2 and F_3 is a bivariate matrix $(\mathbf{M}(u,v))$ of order $2mn$. The singular set of this matrix corresponds to the intersection curve. The intersection curve was evaluated using our algorithm for a number of cases. Some of them are shown in figs.6 and 7. It takes about $1-3$ seconds to evaluate algebraic curves of degree 100 or more with about three or four components in the domain.

Fig.8 shows intersection of two goblets. Each of them is composed of 72 bicubic Bézier patches. The intersection curve is a piecewise algebraic space curve of degree 324. The overall algorithm initially prunes out non-intersecting pairs of surfaces using bounding boxes. Eventually, it evaluates 57 pairs of intersecting surfaces and the resulting algorithm takes about 12.8 seconds. In all the examples, the Euclidean norm was used and the value of δ was 10^{-5}. We believe that a number of optimization techniques can be incorporated in our implementation to give better results. Unfortunately, there are no existing benchmarks available to test our algorithm. Further, there are very few published performance results on surface intersection algorithms. Interval arithmetic has been used to compute surface intersections. In computing the intersection curve between a bumpy sphere and a cylinder, [Sny92] reports a running time of *several minutes* on a HP9000 series 835 workstation. [BHHL88, BR90] use quadratic transformations to remove singularities from the intersection curve and trace it in higher dimensions. [MD94a] provide efficient algorithms to evaluate zero-dimensional algebraic sets. Methods of [AF88, AM88] evaluate the topological types of plane algebraic curves using exact arithmetic. These algorithms provide very accurate results, but are inefficient for high degree curves.

6 Problem Conditioning

The three most important considerations in the evaluation of algebraic curves are *robustness*, *accuracy* and *efficiency*. There is a clear trade-off between these three, since the larger the robustness enhancement and accuracy computations the slower the execution time of the algorithm. While it is almost impossible to provide an algorithm that can satisfy all of them completely together, one must at least target an algorithm that can provide a good fraction of both in most cases and can be fine-tuned according to the requirements of the application.

In order to guarantee robustness, a general algorithm must be able to determine the *conditioning* of the problem.

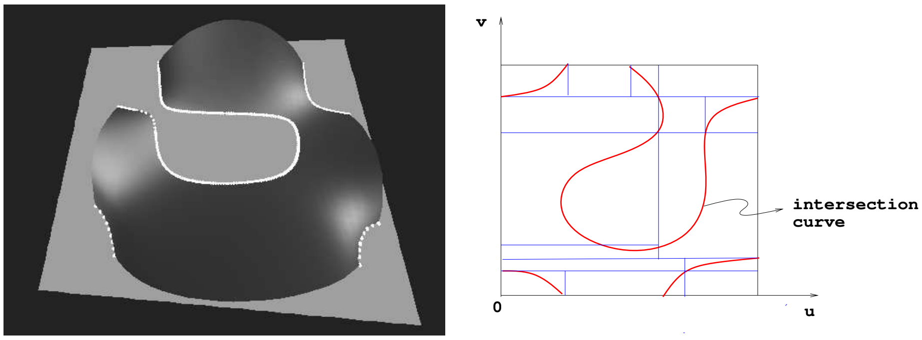

Figure 6: Component Splitting applied to this intersection

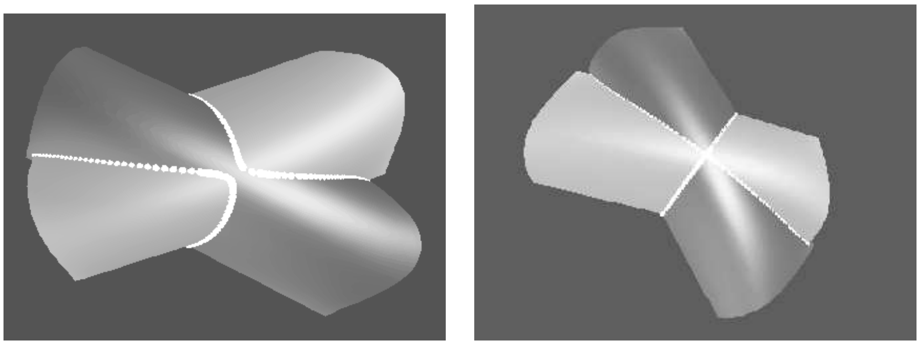

Figure 7: Patches having (i) close intersection curves (ii) intersecting in a singularity

The conditioning becomes more significant because of errors introduced by numerical computations. If the input data changes by ϵ, the output results will change by a function $\delta(\epsilon)$. For very small values of ϵ, there may exist a constant κ such that $\delta(\epsilon) \approx \kappa\epsilon$ [Hof89]. If κ is small the problem is said to be *well-conditioned*. A large value of κ signifies an *ill-conditioned* problem. The value κ is called the *condition number*. However, it is nontrivial to calculate κ for general algebraic problems, as shown by [Ren94]. In our case, the algorithm works well for well-conditioned inputs.

Our algorithm (which reduces starting point computation and tracing to solving eigensystems) can encounter ill-conditioned inputs if the system has higher multiplicity eigenvalues. Since the reduction process involves rounding errors, the resulting system will have a cluster of eigenvalues near the original multiple eigenvalue. However, it has been shown that the *arithmetic mean* of the cluster is usually much less sensitive to small perturbations than individual eigenvalues [Kat80]. In such cases, our algorithm computes the arithmetic mean of the cluster by following a simple heuristic which monitors the condition number of individual eigenvalues [MD94b].

7 Conclusion

We have presented a general algorithm based on numeric and symbolic methods for evaluation of algebraic curves in arbitrary dimensions. It computes a birationally equivalent plane curve using resultants and represents it as a matrix polynomial. The resulting algorithm is based on numerical matrix computations and geometric characterization of the curve. It has been used to accurately compute intersection of high degree surfaces. In terms of efficiency it outperforms earlier methods based on CAD, desingularization etc. and it is more robust and accurate (in terms of identifying all components and singularities) as compared to specific algorithms for surface intersection presented in [Hof89, Hoh91, SN91].

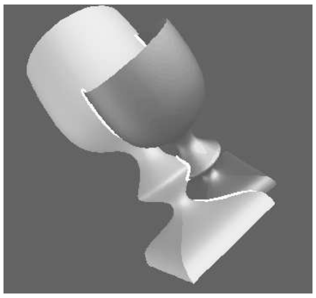

Figure 8: Intersecting goblets

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.

[Abh90] S. S. Abhyankar. *Algebraic Geometry for Scientists and Engineers*. American Mathematical Society, Providence, R. I., 1990.

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

[AM88] D. Arnon and S. McCallum. A polynomial time algorithm for the topological type of a real algebraic curve. *Journal of Symbolic Computation*, 5:213–236, 1988.

[Arn83] D. S. Arnon. Topologically reliable display of algebraic curves. *Computer Graphics*, 17:219–227, 1983.

[AS86] W. Auzinger and H.J. Stetter. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In *International Series of Numerical Mathematics*, volume 86, pages 11–30, 1986.

[BHHL88] C. L. Bajaj, C. M. Hoffmann, J. E. Hopcroft, and R. E. Lynch. Tracing surface intersections. *Computer Aided Geometric Design*, 5:285–307, 1988.

[BR90] C. Bajaj and A. Royappa. The GANITH algebraic geometry toolkit. In *Lecture Notes in Computer Science*, volume 429, pages 268–269. Springer Verlag, 1990.

[Buc89] B. Buchberger. Applications of groebner bases in non–linear computational geometry. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 415–447. MIT Press, 1989.

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

[CE93] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In *Proceedings of AAECC*, pages 89–104. Springer-Verlag, 1993.

[CH] G.W. Crippen and T.F. Havel. Distance geometry and molecular conformation. Research Studies Press, New York.

[Col75] G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Lecture Notes in Computer Science*, number 33, Springer-Verlag, 1975.

[Cra89] J.J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison–Wesley Publishing Company, 1989.

[Dix08] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proceedings of London Mathematical Society*, 6:49–69, 209–236, 1908.

[GBDM77] B.S. Garbow, J.M. Boyle, J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51. Springer-Verlag, Berlin, 1977.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins Press, Baltimore, 1989.

[GLR82] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix Polynomials*. Academic Press, New York, 1982.

[GZ79] C.B. Garcia and W.I. Zangwill. Finding all solutions to polynomial systems and other systems of equations. *Math. Prog.*, 16:159–176, 1979.

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

[Hof90] C.M. Hoffmann. A dimensionality paradigm for surface interrogations. *Computer Aided Geometric Design*, 7:517–532, 1990.

[Hoh91] M.E. Hohmeyer. A surface intersection algorithm based on loop detection. *International Journal of Computational Geometry and Applications*, 1(4):473–490, 1991. Special issue on Solid Modeling.

[Jou91] Jean-Pierre Jouanolou. *Le Formalisme du Résultant*, volume 90 of *Advances in Mathematics*. 1991.

[Kat80] T. Kato. *Perturbation Theory for Linear Operators*. Springer Verlag, Berlin, 2 edition, 1980.

[Laz83] D. Lazard. Groebner bases, gaussian elimination and resolution of systems of algebraic equations. In *EUROCAL'83*. European Computer Algebra Conference, Springer–Verlag, 1983.

[Mac02] F.S. Macaulay. On some formula in elimination. *Proceedings of London Mathematical Society*, 1(33):3–27, May 1902.

[Man92] D. Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics*. PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[Man94] D. Manocha. Computing selected solutions of polynomial equations. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, pages 1–8, Oxford, England, 1994. ACM Press.

[MC27] F. Morley and A.B. Coble. New results in elimination. *American Journal of Mathematics*, 49:463–488, 1927.

[MC91] D. Manocha and J.F. Canny. A new approach for surface intersection. *International Journal of Computational Geometry and Applications*, 1(4):491–516, 1991. Special issue on Solid Modeling.

[MD94a] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves i: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, 1994.

[MD94b] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves ii: multiple intersections. *Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing*, 1994. To appear.

[Moo79] R.E. Moore. *Methods and applications of interval analysis*. SIAM studies in applied mathematics. Siam, 1979.

[Mor92] A. P. Morgan. Polynomial continuation and its relationship to the symbolic reduction of polynomial systems. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 23–45, 1992.

[PK92] J. Ponce and D.J. Kriegman. Elimination theory and computer vision: Recognition and positioning of curved 3d objects from range, intensity, or contours. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 123–146, 1992.

[Ren94] J. Renegar. Is it possible to know a problem instance is ill-posed? *Journal of Complexity*, 10(1):1–56, 1994.

[SN91] T.W. Sederberg and T. Nishita. Geometric hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

[Sny92] J. Snyder. Interval arithmetic for computer graphics. In *Proceedings of ACM Siggraph*, pages 121–130, 1992.

[SS83] J. T. Schwartz and M. Sharir. On the piano movers probelem ii, general techniques for computing topological properties of real algebraic manifolds. *Advances of Applied Maths*, 4:298–351, 1983.

[SZ94] B. Sturmfels and A. Zelevinsky. Multigraded resultants of sylvester type. *Journal of Algebra*, 1994. To appear.

[Wal50] R.J. Walker. *Algebraic Curves*. Princeton University Press, New Jersey, 1950.