An Efficient Surface Intersection Algorithm based on Lower Dimensional Formulation*

Shankar Krishnan
krishnas@cs.unc.edu

Dinesh Manocha
manocha@cs.unc.edu

November 21, 1994

Department of Computer Science,
University of North Carolina,
Chapel Hill, NC 27599-3175

*Supported in part by a junior faculty award, university research council grant, NSF grant CCR-9319957, ONR contract N00014-94-1-0738, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization and NSF Prime contract No. 8920219

Abstract

We present an efficient algorithm to compute the intersection of algebraic and NURBS surfaces. Our approach is based on combining the marching methods with the algebraic formulation. In particular, we use a matrix representation of the intersection curve and compute it accurately using matrix computations. We also present algorithms to compute a start point on each component of the intersection curve (both open and closed components), detect the presence of singularities, find all the curve branches near the singularity, and also suggest methods to compute the step size during tracing to prevent component jumping. The algorithm runs an order of magnitude faster than previously known robust algorithms. The complexity of the algorithm is output sensitive.

1 Introduction

Evaluating the intersection of parametric and algebraic surfaces is a recurring operation in computer graphics, geometric and solid modeling, and computer–aided design. An efficient surface intersection algorithm that is numerically reliable, accepts general surface models, and operates without human supervision is critical to many solid modeling applications. Computation of intersections between surfaces is also required in automatic finite–element mesh generation of three–dimensional solids, simulation of manufacturing processes like tool path generation, interference and feature detection, hidden surface removal, visibility computations and scientific visualization. Intersection of algebraic and NURBS (**N**on-**U**niform **R**ational **B**–**S**plines) surfaces has been listed as one of the most fundamental problem in the integration of geometric and solid modeling systems [Hof89, RR92].

The intersection of two surfaces can be complicated in general, with a number of closed loops and self–intersections (singularities). A good surface intersection algorithm, should, in theory, be able to detect all such features of the intersection curve and trace them correctly in an efficient manner. Surface intersection has been an active area of research for more than three decades. The different approaches can be categorized into subdivision, lattice evaluation, analytic, and marching methods. However, none of them are able to balance three conflicting goals of accuracy, robustness and efficiency [Hof89, Pra86, RR92]. The *robustness* of the algorithm refers to the detection of all curve segments, closed loops and singularities assuming no numerical errors. The surface intersection problem gets further complicated due to numerical errors present in all finite-precision computations [Hof89]. The accuracy characterizes numerical stability of the algorithm in the context of floating point arithmetic. Efficient intersection algorithms are required because they are applied frequently during the design process. As a result, many of the commercial CAD (**C**omputer **A**ided **D**esign) systems, like CATIA, use polyhedral approximations for intersection and boundary computations. These methods suffer from incorrect results and data proliferation. Therefore, it is desirable to design an algorithm that is as robust and efficient as possible, minimizing the potential for missed loops and other curve features which plague many of the present schemes.

Main Result: We present algorithms for efficient computation of surface intersection. Depending on the application requirement, the algorithm can be fine-tuned to provide better robustness guarantees at the expense of execution time. The algorithm has been implemented using double-precision arithmetic. Given combinations of NURBS surfaces, the algorithm decomposes them into Bézier patches. It uses spatial techniques like bounding–box tests and linear programming to reduce the number of pairwise intersections. Given a pair of Bézier and/or algebraic surfaces, it finds a starting point on each component of the intersection curve, decomposes the domain, marches along the curve choosing appropriate step sizes to prevent component jumping, finds the singularities and all branches at each singularity. It has been applied to the intersection of high-degree parametric surfaces (bicubic patches) and performs well. It typically takes about $8 - 10$ seconds on models consisting of about 100 bicubic Bézier surfaces on a SGI Onyx workstation.

1.1 Previous Work

There is a significant body of literature addressing the surface intersection problem. Some recent surveys include [Pat93, Pra86, Hof89]. Surface intersection algorithms can be broadly classified into four major categories: *subdivision, lattice evaluation, analytic and marching methods*. More recently, techniques have been designed that combine features of different categories and are generally referred to as *hybrid methods*.

Subdivision methods: The basic idea of these methods is to decompose the problem recursively into similar problems which are much simpler. Decomposition continues until a desired level of simplicity is achieved and then the corresponding intersection is obtained directly. The last step is to merge all the individual curves together to get the final solution. This approach has the flavor of the *divide and conquer* paradigm extensively used in algorithmic design. Subdivisions are based on the geometric properties of the control polytopes [LR80, Gei83, Las86]. These methods are convergent in the limit but if used for high-precision results lead to data proliferation and are consequently slow. In case subdivision is stopped at some finite steps, it may miss small loops or lead to incorrect connectivity in the presence of singularities. The robustness of this approach can be improved by posing the problem algebraically and using *interval arithmetic* [Sny92]. Its performance is too slow for most applications.

Lattice evaluation: These techniques decompose surface intersection into a series of lower geometric complexity problems like curve-surface intersections [RR87]. This is followed by connecting these discrete points into curves. Determination of the discrete step size to guarantee robust solutions is hard. Further, they can be slow and suffer from robustness problems in terms of finding all the small loops and singularities.

Analytic methods: Analytic methods are based on explicit representation of the intersection curve and have been restricted to low degree intersections [Sed83, Sar83]. Another alternative to the analytic methods is the use of geometric methods developed by [Piegl89]. In this paper, Piegl uses geometric principles to compute the intersection of quadric surfaces and are very accurate. However, the algorithm cannot be easily extended to the general intersection problem.

Marching methods: These are by far the most widely used [Far86, BHHL88, BK90, KPW90] because of their generality and ease of implementation. The basic advantage of this technique is its generality, allowing intersection of arbitrary parametric surfaces like offsets and blends. The idea behind marching methods involve analytic formulation of the intersection curve, determination of a start point on each component, and the use of local geometry to trace out the curve. The intersection curve can be defined implicitly as an algebraic set based on the surface equations or as a curve of zero distance between the two surfaces or as a vector field [Hof90, KPW90, Che89]. Tracing can be done on the intersection curve in higher dimensions defined by its variables or on its projection in the plane. According to [Hof90], the projection results in a high-degree formulation and its computation can be inefficient and numerically unreliable. To circumvent these problems, a representation based on *unevaluated determinant* was introduced in [MC91].

The components of an intersection curve consist of boundary segments and closed loops. Start points on the boundary segments are obtained by curve-surface intersections [SN91]. Many techniques have appeared over the last few years to detect closed loops on the intersection curve [SM88, Hoh91, KPP90, ZS93]. They are based on bounds on *gauss maps* and subdivide each surface until sufficient conditions for the non-existence of loops are satisfied. These algorithms work very well to isolate cases with no loops only if the surfaces are relatively flat. However, in the presence of small loops or singularities, the algorithm becomes slow.

Most algorithms use the local geometry of the curve coupled with quasi-Newton's methods [BHHL88, BK90] for tracing. These methods do not converge well sometimes [FF92] and many issues related to choice of step size to prevent component jumping are still open. Therefore, most implementations use very conservative step sizes for tracing and this slows down the algorithm. Overall, current tracing algorithms are not considered robust [Sny92].

The *singularities* on the intersection curve can be classified in terms of solutions of algebraic equations. However, no methods are known in the literature which can efficiently compute them for high degree surfaces (like bicubic patches) and classify the curve branches in its neighborhood.

1.2 Overview of our Algorithm

Our algorithm formulates the intersection problem algebraically, computes the projection of the intersection curve as an algebraic plane curve and evaluates it. The projection is represented as the singular set of a bivariate matrix polynomial and the algorithm uses matrix computations to trace the curve. Tracing is a geometric operation and evaluating the curve in the plane reduces its *geometric complexity*. The loops of the intersection curve are characterized algebraically and computed using curve-surface intersections followed by complex tracing. The tracing algorithm employs *domain decomposition* and use of inverse power iterations. Domain decomposition provides a natural procedure to compute the step size and prevent *component jumping*. This is a major advantage of our tracing algorithm since incorrect connectivity of the various curve components was considered a flaw with traditional marching methods. We also show that we need not require the computation of singular points before the tracing step. Our algorithm can detect the presence of singularities during the tracing process. The accuracy of the result is improved by performing minimization on a distance-based energy function. To speed up the algorithm we also use existing simple geometric tests to isolate cases consisting of no loops or singularities. Fig. 1 shows the intersections of two teapots (Utah teapots) and fig. 2 shows the intersection curve in space between the two teapots. Each teapot is composed of 32 Bézier patches. In all the figures

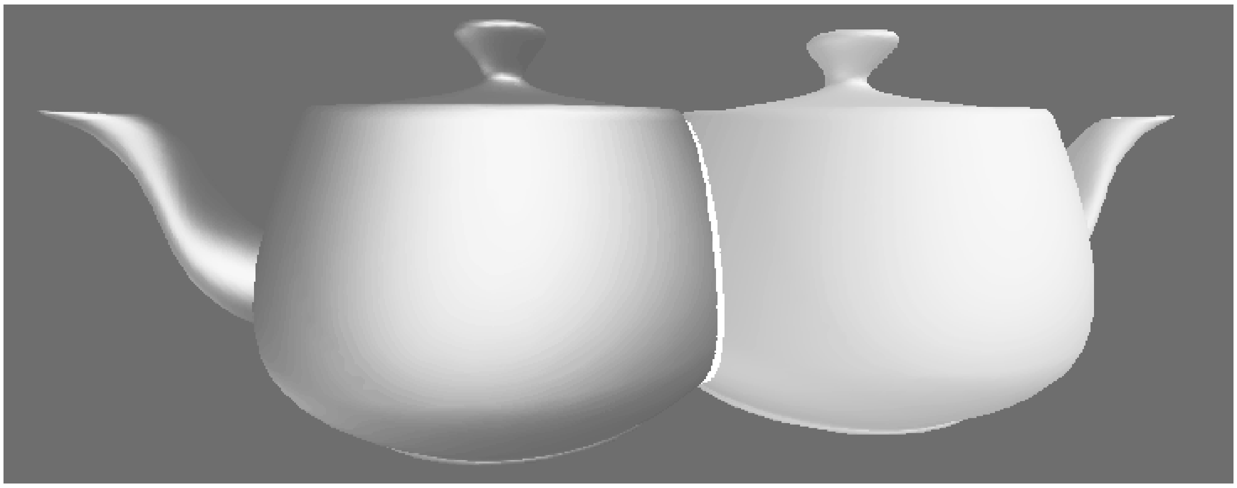

Figure 1: Intersection of the Utah Teapots

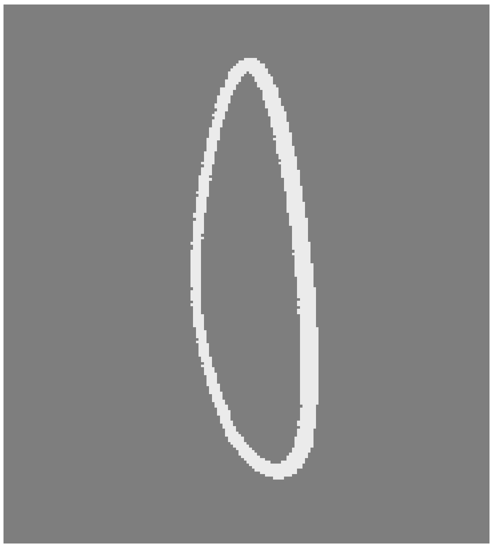

Figure 2: Intersection curve of figure 1

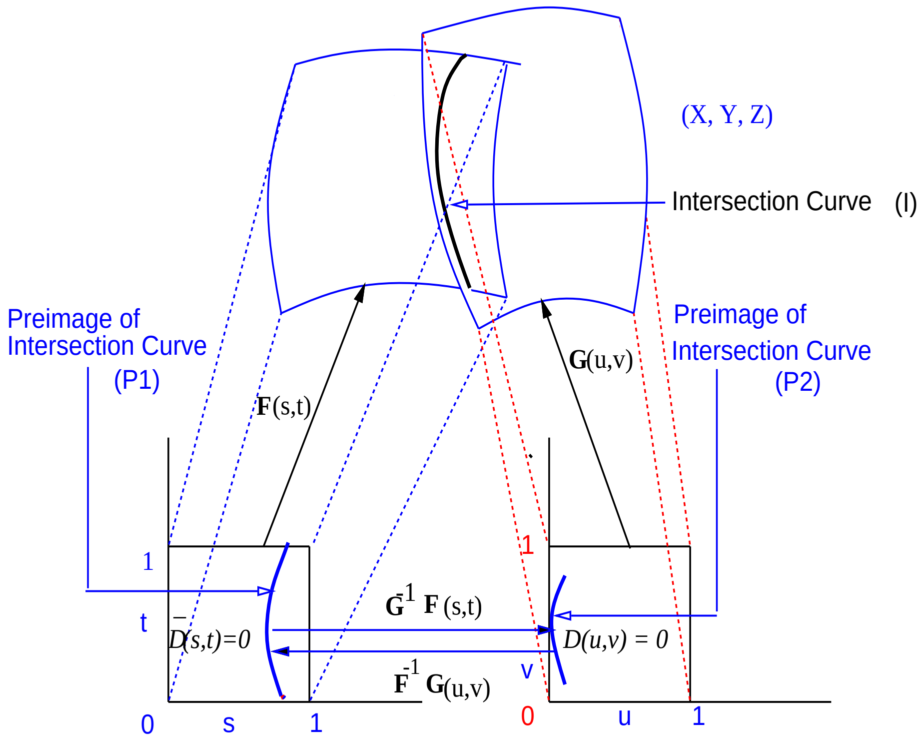

Figure 3: Intersection curve and its planar preimages [MC91]

displaying intersection curves, we have used spheres of a small constant radius to represent the curve clearly as traced out by the algorithm. Because of perspective projection, some of the spheres appear much larger than the other.

The rest of the paper is organized in the following manner. We review the problem formulation and the representation of the intersection curve in terms of matrices in section 2. Section 3 describes algorithms to compute start points on every curve component including loops. The tracing algorithm using domain decomposition is presented in section 4. Methods for identification of singularities on the intersection curve are given both in section 4 and 5. Section 6 outlines the main issues concerning robustness and efficiency, and the contributions of our algorithm. Application of our intersection algorithm on large-sized models are presented in section 7. This is followed by a dicussion on the performance of our algorithm on these models. Finally we identify some algorithmic and implementation issues that are still open.

2 Background

Most of the commercial CAD systems use tensor-product surfaces to represent their models, though recently triangular surfaces and other representations are gaining importance as well. Initially we present the algorithm for computing the intersection of two general Bézier surfaces. Later we generalize them to compute the intersection of NURBS models. We present the intersection algorithm for parametric surfaces. The algorithm can be directly applied to implicit algebraic surfaces as well. The problem of surface intersection corresponds to computing an accurate representation of the intersection curve. The difficulty of the problem lies both in the *algebraic* and the *geometric complexity* of the intersection curve. The degree of the curve resulting from the intersection of rational parametric surfaces can be very high, and computing an accurate representation is nontrivial. For example, the degree of the intersection curve of two tensor-product bicubic Bézier surfaces can be as high as 324. In terms of geometric complexities, the curve may have multiple components, small loops, singularities and multiple branches at the singularities. Our approach is based on the exact representation of the intersection curve. It is a well known result in algebraic geometry that the intersection space curve has a one-to-one correspondence with an algebraic plane curve (except for a finite number of points). The plane curves with one-to-one correspondence with the intersection curve in space are shown in fig. 3. We represent the plane curve as an unevaluated determinant [MC91].

Matrix Formulation: In this paper, we shall assume that the parametric surface is given in the form of a *tensor product Bézier patch.* A tensor product patch is of the form

$$F(s,t) = (\sum_{i=0}^{m}\sum_{j=0}^{n} V_{ij} B_{i,m}(s) B_{j,n}(t)),$$

where $V_{ij} = (x_{ij}, y_{ij}, z_{ij}, w_{ij})$ are the control point coordinates and $B_{i,m}(s) = \binom{m}{i} s^i(1-s)^{m-i}$ is the Bernstein polynomial. Many of the algorithms presented in this paper assume polynomials represented in the power basis instead of the Bernstein basis. Therefore, to convert from the Bernstein to the power basis, we perform the domain transformations

$$\overline{s} = g(s) = \frac{s}{(1-s)}, \quad \overline{t} = g(t) = \frac{t}{(1-t).}$$

Given two Bézier surfaces,

$$\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t))$$

$$\mathbf{G}(u,v) = (\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v))$$

in homogeneous coordinates, implicitize $\mathbf{F}(s,t)$ to the form $f(x,y,z,w) = 0$ [Sed83, Hof89] and substitute the parametrization of $\mathbf{G}(u,v)$ into f to get an algebraic plane curve of the form

$$f(\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v)) = 0.$$

This corresponds to an algebraic plane curve birationally equivalent to the original intersection curve. However its degree it rather high and leads to efficiency and accuracy problems [Hof89].

The implicit representation of the patch is obtained by eliminating s and t from

$$xW(s,t) - X(s,t) = 0$$

$$yW(s,t) - Y(s,t) = 0$$

$$zW(s,t) - Z(s,t) = 0$$

using *resultants* [Sed83]. There are different formulations of resultants for tensor product surfaces and triangular surfaces. It turns out that the resultant of these three equations can always be expressed as the determinant of a matrix [Dix08]. Let us denote that matrix as $\overline{\mathbf{M}}(x,y,z,w)$. Furthermore, each entry of the matrix is of the form $a_{ij}x + b_{ij}y + c_{ij}z + d_{ij}w$. The order of $\overline{\mathbf{M}}(x,y,z,w)$ is a function of the degrees of the equations. In particular, for tensor product surfaces of the form $s^m t^n$, the order of the matrix is $2mn$ and for triangular surfaces of degree n the order is $2n^2 - n$. The determinant of the resulting matrix corresponds to the implicit representation of the parametric surface. We substitute the parametrization of $\mathbf{G}(u,v)$ into this matrix and obtain a representation of the form $\mathbf{M}(u,v)$, where each entry is a polynomial in u and v. This substitution is very simple because every entry of the matrix is just a linear term. The degree of each polynomial corresponds to the degree of $\mathbf{G}(u,v)$.

Base Points: The base points of a parametrization are the common solutions of the equations: $X(s,t) = 0$, $Y(s,t) = 0$, $Z(s,t) = 0$, $W(s,t) = 0$. Typically, a generic parametrization does not have base points. However, in the presence of base points the resultant of the parametric equations is identically zero. The implicit representation in such cases is a factor of the determinant of the maximal non–vanishing minor of $\mathbf{M}(x,y,z,w)$ [MC91]. We use that minor for the representation and substitute the parametrization of $\mathbf{G}(u,v)$ to obtain the planar projection of the intersection curve. We denote this matrix also as $\mathbf{M}(u,v)$. However, the determinant of $\mathbf{M}(u,v)$ may consist of extraneous components. We handle patches with base points specially. The details are presented in [SM94].

Matrix Computations: 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 intersection 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 respectively at (u_1,v_1). The resulting matrix structure is of the form

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

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 kth row and jth column of the matrix. In the new formulation this step is replaced as

$$\begin{aligned} g_{kj} &= g_{kj} - \frac{g_{ki}}{g_{ii}} g_{ij}, \\ g^u_{kj} &= g^u_{kj} - \frac{(g^u_{ki} g_{ij} + g_{ki} g^u_{ij}) g_{ii} - (g_{ki} g_{ij}) g^u_{ii}}{(g_{ii})^2} \\ g^v_{kj} &= g^v_{kj} - \frac{(g^v_{ki} g_{ij} + g_{ki} g^v_{ij}) g_{ii} - (g_{ki} g_{ij}) g^v_{ii}}{(g_{ii})^2} \end{aligned} \tag{1}$$

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

$$D(u_1,v_1) = \prod_{i=1}^{n} g_{ii}$$

$$D^u(u_1,v_1) = D(u_1,v_1) \sum_{i=1}^{n} \frac{g^u_{ii}}{g_{ii}}.$$

$$D^v(u_1,v_1) = D(u_1,v_1) \sum_{i=1}^{n} \frac{g^v_{ii}}{g_{ii}}.$$

This procedure can be easily extended to compute the higher order partial derivatives as well. Furthermore, the analysis of Gaussian elimination may be used for analyzing the numerical accuracy of partial derivatives computation.

3 Intersection Computation

The intersection curve in the domain of $\mathbf{G}(u, v)$ is defined as the *singular set* of the matrix polynomial $\mathbf{M}(u, v)$. In other words, it consists of all points $(u_1, v_1) \in [0, 1] \times [0, 1]$ such that $\mathbf{M}(u_1, v_1)$ is singular. Corresponding to each point (u_1, v_1) there exists a point $(s_1, t_1) \in [0, 1] \times [0, 1]$ in the domain of $\mathbf{F}(s, t)$. Given a point (u_1, v_1) in the domain of $\mathbf{G}(u, v)$, (s_1, t_1) can be computed from a vector in the *kernel* of $\mathbf{M}(u_1, v_1)$ [Dix08]. The main advantages of this matrix representation are its efficiency and accuracy. Although the singular set is defined in terms of a determinant, we use algorithms based on *eigenvalues* and *singular values* for numerical stability. Efficient and accurate algorithms for computing the eigendecomposition and SVD (**S**ingular **V**alue **D**ecomposition) are well known [GL89] and good implementations are available as part of numerical libraries like EISPACK and LAPACK.

3.1 Curve-Surface Intersections

The algorithms for computing start points on the intersection curve and decomposing the intersection curve for robust tracing are based on Bézier curve-surface intersections. There are many algorithms known for computing these intersections based on subdivision properties of control polytopes. Their convergence can be improved by Bézier clipping [SN91]. We use some recent algorithms for these intersections based on eigenvalue computations [Man93]. Given a surface $\mathbf{F}(s, t)$, we compute its implicit representation as described in the previous section and obtain a matrix formulation $\overline{\mathbf{M}}(x, y, z, w)$. We substitute the parametrization of the curve, say $\mathbf{G}(u) = (\overline{X}(u), \overline{Y}(u), \overline{Z}(u), \overline{W}(u))$ of degree d, and obtain a univariate matrix polynomial $\mathbf{M}(u)$. The problem of intersection computation reduces to computing the roots of the matrix polynomial $\mathbf{M}(u)$. Each entry of the matrix polynomial is expressed in Bernstein basis. We divide it by $(1-u)^d$ and use a reparametrization of the form $\overline{u} = \frac{u}{1-u}$ to obtain a representation in the power basis. Direct conversion from Bernstein to power basis can introduce numerical problems [FR87]. The resulting matrix $\mathbf{M}(\overline{u})$ can be represented as

$$\mathbf{M}(\overline{u}) = \overline{u}^d M_d + \overline{u}^{d-1} M_{d-1} + \ldots + \overline{u} M_1 + M_0. \tag{2}$$

where M_i's are matrices of order $2mn$ with numeric entries. Furthermore, the roots of the matrix polynomial, $\mathbf{M}(u)$, have one-to-one correspondence with the eigendecomposition of

$$C = \begin{bmatrix} 0 & I_n & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & I_n \\ -\overline{\mathbf{M}}_0 & -\overline{\mathbf{M}}_1 & -\overline{\mathbf{M}}_2 & \ldots & -\overline{\mathbf{M}}_{d-1} \end{bmatrix} \tag{3}$$

where $\overline{\mathbf{M}}_i = \mathbf{M}_d^{-1}\mathbf{M}_i$ [GLR82]. In case $\mathbf{M}_d$ is singular or ill-conditioned, the intersection problem is reduced to a generalized eigenvalue problem [Man93]. Algorithms to compute all the eigenvalues are based on QR orthogonal transformations [GL89]. They compute all the real as well as complex eigenvalues. Algorithms to compute eigenvalues in a subset of the real or complex domain are presented in [M93], based on *algebraic pruning*. If there are few intersections (two or three in the domain), the pruning approach is about an order of magnitude faster than the QR algorithm.

3.2 Start Points

The marching algorithm needs start points on each component of the intersection curve. These components can be classified into *open* and *closed* components. Open components have an intersection with one of the boundary curves of the surface as shown in fig.3. These are points on the plane curve where one of the parameter values is 0 or 1. The other components are closed loops. The start points on the open components are computed using curve-surface intersections. In particular, we substitute $u = 0, u = 1, v = 0$ and $v = 1$ in the representation of the intersection curve, $\mathbf{M}(u, v)$, and compute the intersections by algebraic pruning.

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 some point on every loop. We shall describe our idea of loop detection in detail next.

3.3 Loop Detection

A number of algorithms have appeared on finding loops in surface intersection [SM88, Hoh91, KPP90, ZS93]. They are based on geometric properties of the surfaces and arrive at a sufficient criterion for two surfaces to have no loop intersections. The first such condition was developed by Sinha [SKW85]. It was proved that two surface patches cannot intersect in a loop if their corresponding *gauss maps* do not overlap. Gauss maps are basically the projections of the normal vector field of the surface on a unit sphere. Techniques for approximating the gauss map using bounding cones was introduced in [SM88].

Hohmeyer [Hoh91] reduced the problem of detecting overlap of gauss maps to a linear programming problem. A pseudo-normal patch ($\mathbf{F}_s \times \mathbf{F}_t$) is evaluated for each parametric surface as another Bézier patch and the control points are used to check for a separating plane. Linear programming can be done in linear time (in fixed dimensions), and some very fast practical algorithms have been designed [Sei90]. If the two surfaces do not satisfy the test, the patches are subdivided and the algorithm is applied recursively. The algorithms of [Hoh91, ZS93] work very well in practice in isolating intersections with no loops only if the surfaces are relatively flat. If the intersection curve has a singularity or a small loop, it may take too many levels of subdivision to detect them. Furthermore, the loop detection criterion are based on the normals of the patches and their computation is expensive for rational patches at every level of subdivision. The normal patch for a rational patch of degree m (in u) and n (in v) is of degree $3m$ (in u) and $3n$ (in v).For example, eight levels of subdivision were required for the intersection of the two patches shown in fig.4.

We initially use Hohmeyer's algorithm based on Gauss maps and linear programming to

Figure 4: Two surfaces intersecting in a loop

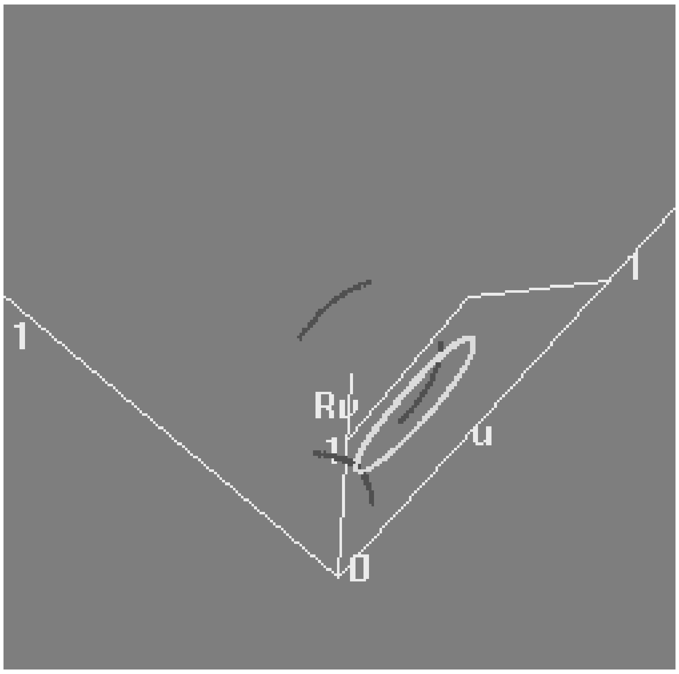

Figure 5: Tracing in complex space

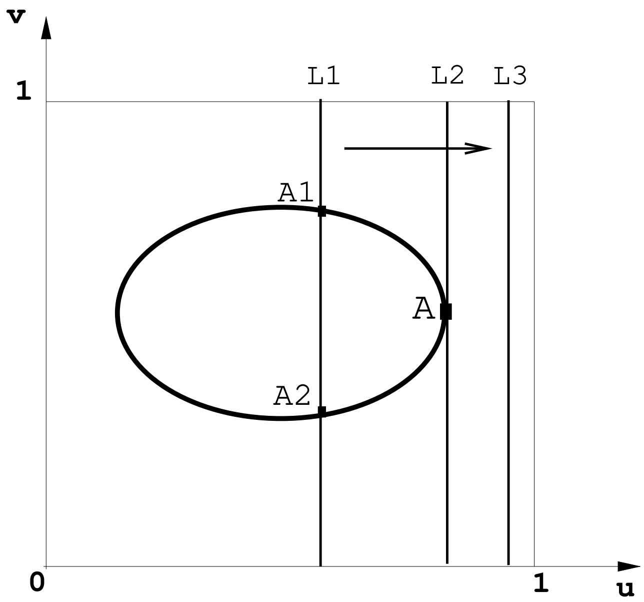

Figure 6: Characterization of loops

check for absence of loops [Hoh91]. If the loop detection criterion is not satisfied, we use the algorithm presented below for computing start points on the loops.

The intersection curve 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 $(u, v) \in [0,1] \times [0,1]$. 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, some basic definitions have to be introduced.

Definition 1 Turning points *are points on the intersection curve where the tangent vector, as projected in the (u, v) space, is parallel to the u or v parameter axes. In other words, one of the partial derivatives (with respect to u or v) of the intersection 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 ($u = u_1 + \delta$).*

Definition 2 Isoparametric curves *are curves lying on a parametric patch (surface) where one of the parameters of the patch (u or v) remains constant.*

Lemma 1 *If the intersection curve in the real domain $[0,1] \times [0,1]$ 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 based on *Bezout's theorem* which states that *if f and g are two algebraic curves of degree m and n respectively, then f and g intersect in exactly mn points in the complex domain counted properly, or they have a common component.* Using Bezout's theorem and the fact that the intersection curve forms a continuous set in the complex domain, the lemma can be proved.

Let us consider two surfaces that intersect in a loop in the real domain, like the ones shown in fig.4. All isoparametric curves on a surface have the same degree, namely the degree of the other parameter defining the surface. Therefore, the number of intersections of the intersection curve with any isoparametric curve must be fixed in complex space (the Bezout bound). Fig.6 illustrates the argument. The line $L1$ intersects the intersection curve at two different real points. As we move the line from $L1$ to $L2$, the two intersection points come closer, and at line $L2$, both of them coincide to form a double root maintaining the intersection count constant. This double root also corresponds to a u-turning point. At $L3$ there are no real intersections. Since the intersection curve is continuous in complex domain, the double root must now be complex values and occur in conjugate pairs (real algebraic curve). Now if the sweep is started from $L3$ towards $L2$, the complex conjugate components come closer together, and at line $L2$, their imaginary part vanishes to yield a double root.

The double root is a real point on the loop (or any other open component with a turning point).

□

The domain of the intersection curve in the complex space is shown in fig.5. The third axis corresponds to the imaginary components of v. It represents a continuous component of the intersection curve. The *dark* curve is the intersection curve in the complex space and the *white* curve is the part of the curve that lies in the real plane.

We need one start point on each loop to trace it. So we restrict ourselves to u-turning points. Henceforth, we shall use *turning points* to denote u-turning points. Our 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. To compute the turning points on the curve, the algorithm combines curve-surface intersections and complex tracing. We could compute the complex solutions of the curve $\mathbf{M}(0, v) = 0$ using eigendecomposition. Unfortunately, these are not the only complex paths that could lead to a turning point. There could be complex paths starting from *right turning points* of some other component of the intersection curve. This can be illustrated by considering the intersection between a bicubic patch and a plane (see figs.7 and 8). The curve $\mathbf{M}(0, v) = 0$ is a cubic curve with two real solutions inside the domain. This implies that there cannot be any complex solution to this equation. Therefore, the left turning point on the loop is connected in complex space to the right turning point of another component. So we use the following strategy to complete a sweep of the complex paths from $u = 0$ to $u = 1$.

Since complex solutions occur in conjugate pairs (real algebraic equation), we restrict ourselves to complex paths whose imaginary parts are strictly positive. When a complex path touches the real plane the imaginary part (of v) must reach 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

$$\det(\mathbf{M_r}(u, v_r) + i\mathbf{M_i}(u, v_r)) = 0 \tag{4}$$

It is easy to show that the solutions (u, v_r) satisfying equation (4) also satisfy the solution of $\det \mathbf{P}(u, v_r) = 0$, where

$$\mathbf{P}(u, v_r) = \begin{bmatrix} \mathbf{M_r}(u, v_r) & -\mathbf{M_i}(u, v_r) \\ \mathbf{M_i}(u, v_r) & \mathbf{M_r}(u, v_r) \end{bmatrix} \tag{5}$$

As before, the solutions to (5) can be posed as the singular set of matrix $\mathbf{P}(u, v_r)$. The singular set of $\mathbf{P}(u, v_r)$ is a discrete point set. The order of the matrix $\mathbf{P}(u, v_r)$ is twice that of $\mathbf{M}(u, v)$. Therefore, there are twice as many paths to trace in general. If the patches are of order $m \times n$ and $p \times q$, then at most $2mnp$ paths have to be traced.

Initially we form the companion matrix of $\mathbf{P}(u, v_r)$, C_p, similar to the one in Eq.(3). We compute all the eigenvalues of C_p at $u = 0$ (we expect all of them to be complex). We use

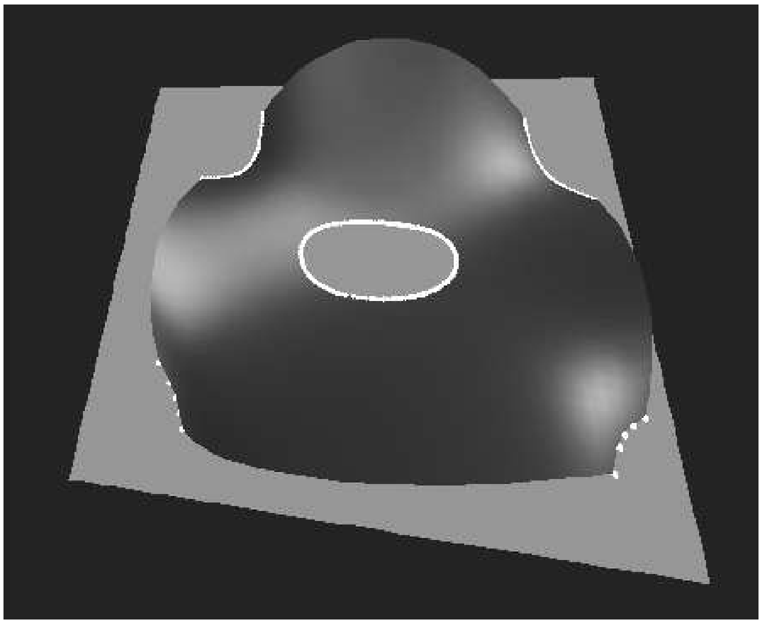

Figure 7: Intersection of a plane with a bicubic patch

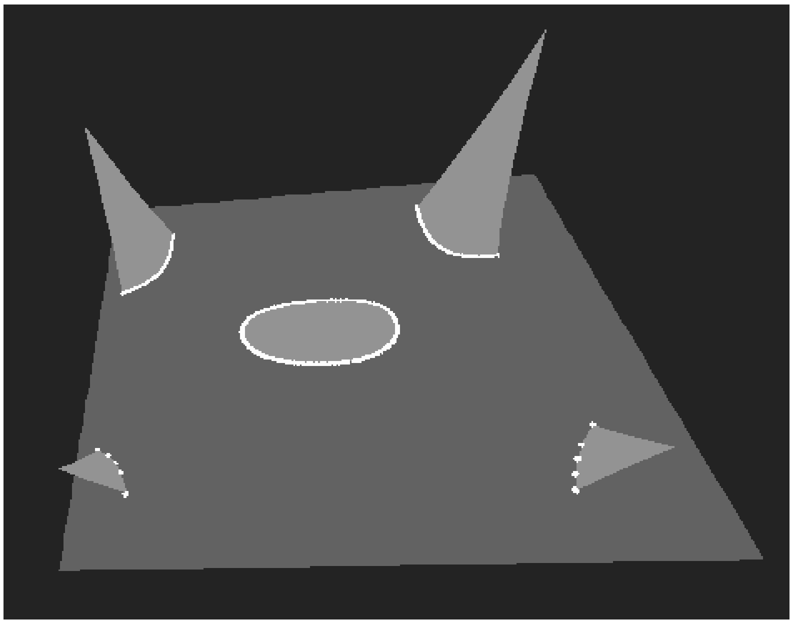

Figure 8: Same intersection shown from another angle

them as starting points and trace all the paths in increasing u direction until it either crosses the $u = 1$ 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 could potentially lead to loops are obtained. The details of tracing using inverse power iterations are presented in the next section. Figs. 7 and 8 show an intersection of a plane with a bicubic patch. The intersection contains a number of open components and a loop. The complex tracing idea was applied here, and a point on the loop was found.

4 Tracing

Given the start points, we evaluate the curve using our tracing algorithm. There are a number of algorithms proposed for tracing [BHHL88, BK90, Che89, KPW90]. 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. Given the approximate value, these algorithms use local iterative methods like Newton's method to trace back on to the curve.

The three main problems with tracing algorithms are [FF92, Sny92]:

1. Converging back on to the curve.

2. Component jumping.

3. Inability to handle singularities and multiple branches.

The convergence problems arising from the behavior of Newton's method are highlighted in [FF92]. It is rather difficult to predict the convergence of Newton's method on high degree equations corresponding to the intersection (for bicubic patches). Component jumping can occur when two components of the curve are relatively close to each other as shown in fig. 9(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. No efficient algorithms are known for handling singularities on the intersection curve of high degree surfaces (say bicubic patches).

We present an efficient tracing algorithm that can resolve all the issues most of the time. In particular we introduce a technique called *domain decomposition* and tracing based on *inverse power iterations*. Singularities are also handled efficiently under well-placed assumptions.

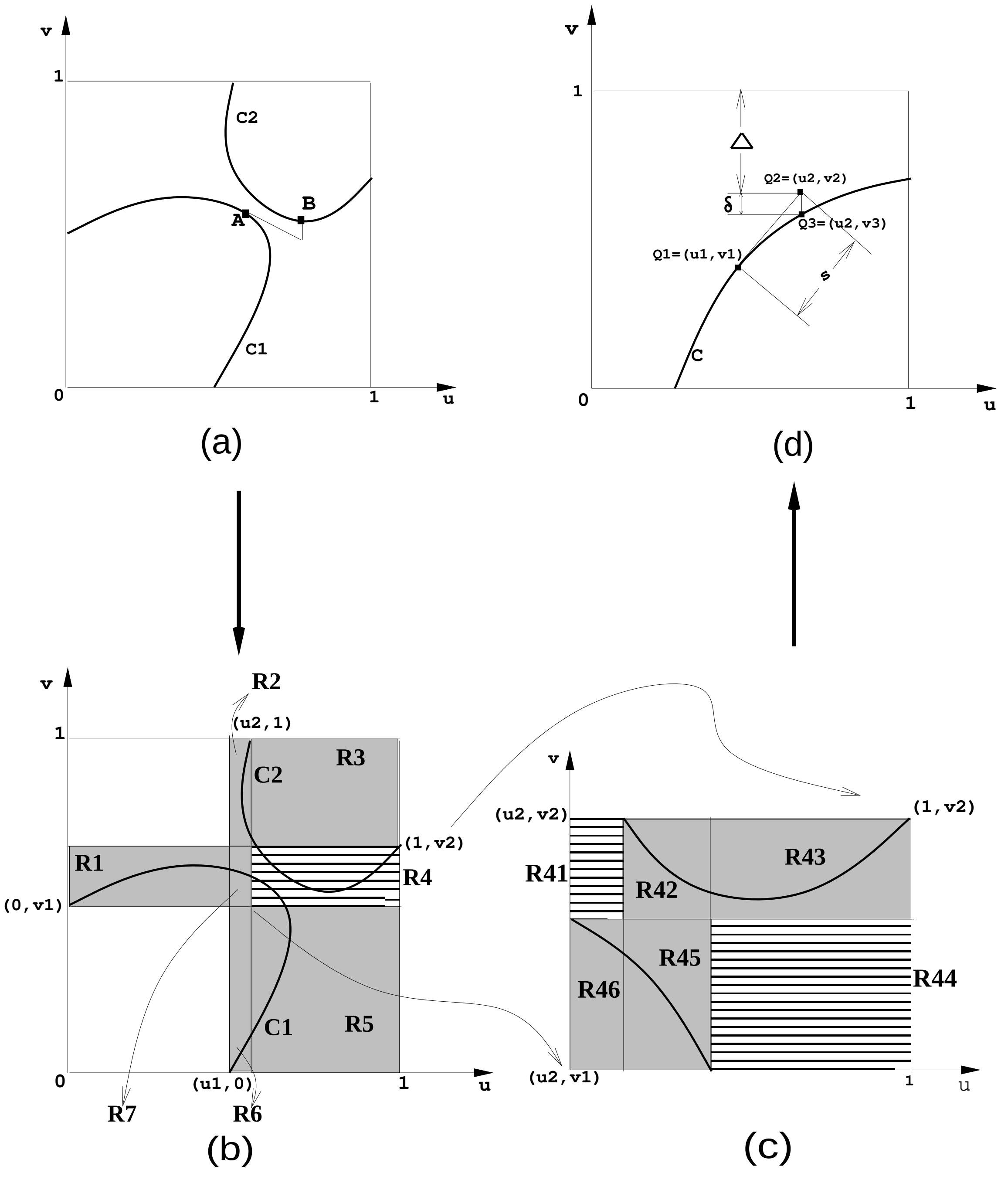

Figure 9: (a)Component jumping (b)First-level decomposition (c)Second-level decomposition (d)Tracing step

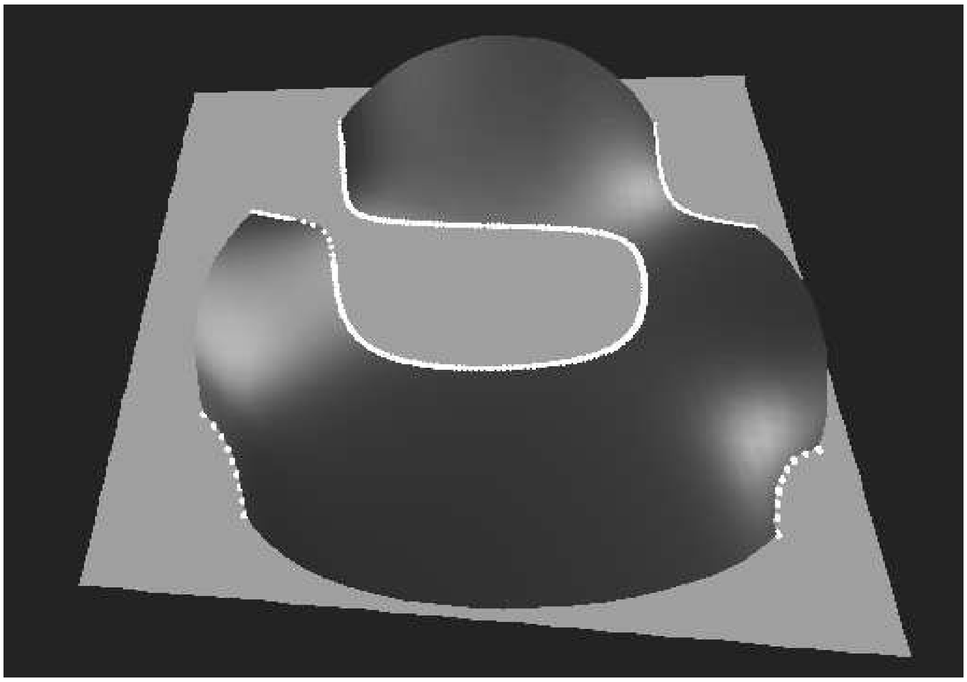

Figure 10: Domain Decomposition applied to this intersection

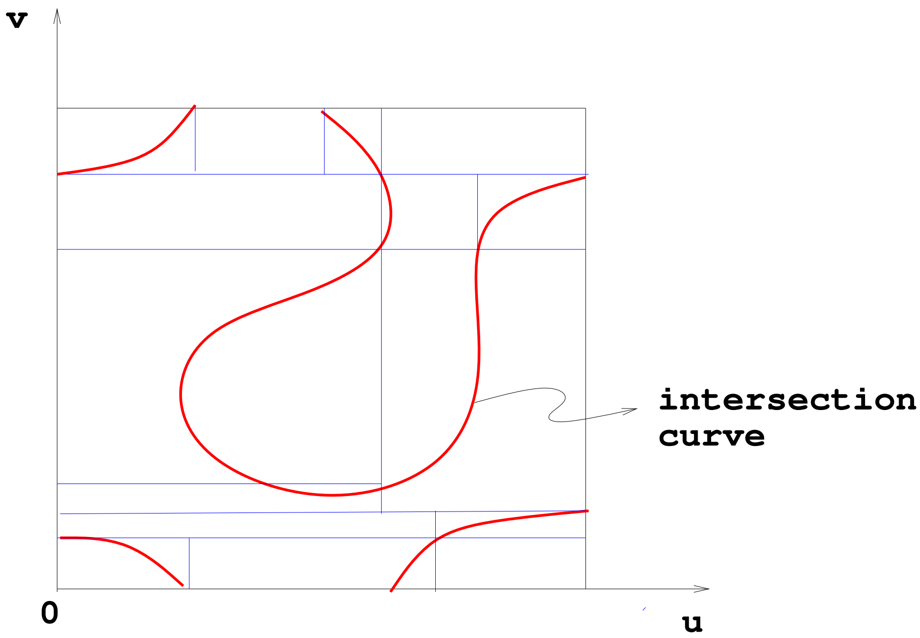

Figure 11: Working of domain decomposition on the intersection of fig.10

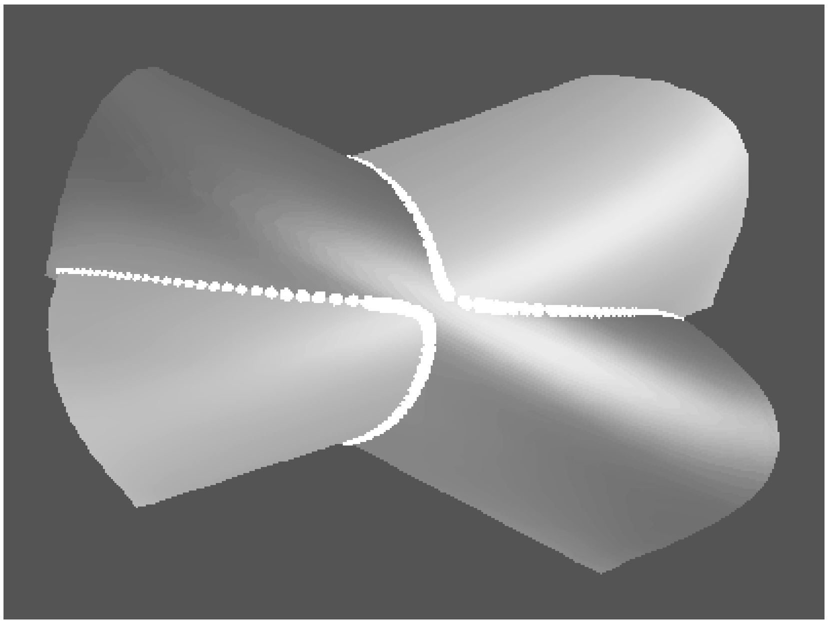

Figure 12: Two patches having very close intersection curves

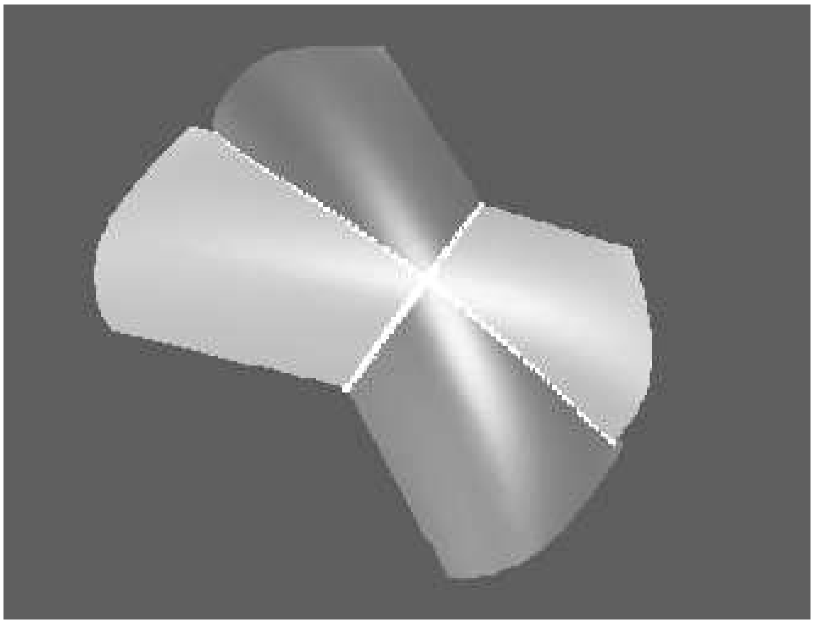

Figure 13: Two patches intersecting in a singularity

4.1 Domain Decomposition

After performing curve-surface intersection and loop detection, a sequence of points are obtained on the curve ($(u, v) \in [0, 1] \times [0, 1]$) which either correspond to starting points on open components or some point on loops. Using these points the intersection curve is traced completely without missing any important curve features. The idea behind *domain decomposition* is that if there are only two boundary points inside a domain with no loops, these points belong to the same component of the intersection curve. Further there exists exactly one component of the intersection curve inside this domain. Therefore, the purpose of the algorithm is to subdivide the original domain into smaller subdomains such that each subdomain contains exactly one curve component.

We now describe the working of domain decomposition. The input into the domain decomposition routine is a rectangular domain, specified as $[L_u, H_u] \times [L_v, H_v]$, and a set of points, S, on the intersection 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 inside the domain and the decomposition terminates.

If the cardinality of S is greater than two the algorithm subdivides the domain along isoparametric lines determined by the parametric values of points of S. The isoparametric lines 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 isoparametric lines with the intersection curve are generated and inserted into the appropriate subdomains. Domain decomposition is then applied recursively to each subdomain.

Subdivision 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 intersection curve where the curve self-intersects. A more formal treatment of singularities is given in the next section. In the presence of singularities (except for cusps), however, no level of decomposition can produce subdomains with one simple curve component unless the singular point is determined accurately. If domain decomposition is unable to isolate single curves in a domain after repeated levels of subdivision, then one of two cases can occur.

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

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

$$E(u, v, s, t) = (D(u, v, s, t)^2 + N(u, v, s, t)^2) \tag{6}$$

where,

$$D(u, v, s, t) = \mid \mathbf{F}(s, t) - \mathbf{G}(u, v) \mid \tag{7}$$

and,

$$N(u,v,s,t) = \mid (\mathbf{F_s}(s,t) \times \mathbf{F_t}(s,t)) \times (\mathbf{G_u}(u,v) \times \mathbf{G_v}(u,v)) \mid \tag{8}$$

$\mid . \mid$ refers to the length operator and $\times$, the cross product of two vectors. The minimization is applied with the midpoint of the region given as the initial point. A minimum value of zero 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 domain decomposition is performed at each subdomain. Singularities are unstable points and are 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.

The pseudocode for the domain decomposition algorithm is as follows:

- **DomainDecomposition**(*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 DomainDecomposition(subregion, new_points, *tolerance*).
       * Return.

       }

     }
  3. If (domain convergence is slow) {
     - Divide the domain at midpoint of one of the parameters.
     - Compute the intersections of the curve with the dividing line.
     - For each of the two subregions, do DomainDecomposition(subregion, new_points, *tolerance*).
     - Return.

     } else {
     - Divide the domain along isoparametric lines 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 DomainDecomposition(subregion, new_points, *tolerance*).
     - Return.

}

}

At the end of the DomainDecomposition algorithm, a set of curves traced out 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 intersection curve in the $[0,1] \times [0,1]$ 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, singular points are always bracketed. It is possible to miss some singular points, however, if they are not well-separated.

The decomposition algorithm has been highlighted on tensor product surfaces in figs.9(b), 9(c) and 11. Typically, the algorithm uses one or two levels of decomposition. However in a few cases, the number of levels may be more. These cases arise only in the presence of singularities or close components. In that case, the geometry of the curve is not simple and imposes a lower bound on the complexity of any robust algorithm. The decomposition step is similar in nature to that of subdivision or interval arithmetic based algorithms. However, subdivision is done only for separating the components and not evaluating them to a certain accuracy. For almost all cases domain decomposition performs fewer levels of decomposition. Subdivision and interval arithmetic based algorithms perform a number of decompositions as a function of the accuracy parameter. The algorithm has been implemented and tested on a wide variety of intersections and it is an order of magnitude faster than previously known robust algorithms (like interval arithmetic).

Figs.10, 12 and 13 show the power of domain decomposition. It can take care of arbitrary intersections. Fig.12 shows the intersection of two patches where the intersection curves come very close to each other. Fig.13 shows the same two patches intersecting in a singularity. Domain decomposition was able to detect the presence of singularity in the second case, and traced all the branches correctly.

4.1.1 Convergence

Fig.14 provides a comparison between ordinary bisection and domain decomposition. It can be seen that in the case depicted by the figure, our method performs much better than bisection. In fact, on an average, domain decomposition 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.

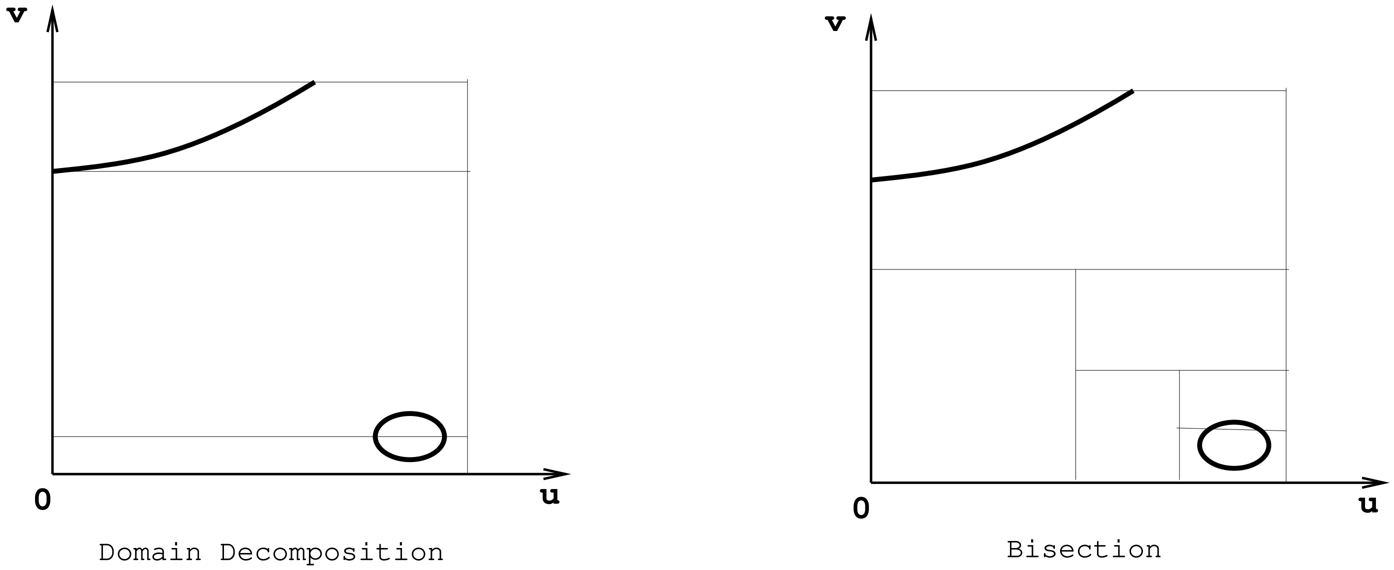

Figure 14: Comparing Domain Decomposition and Bisection

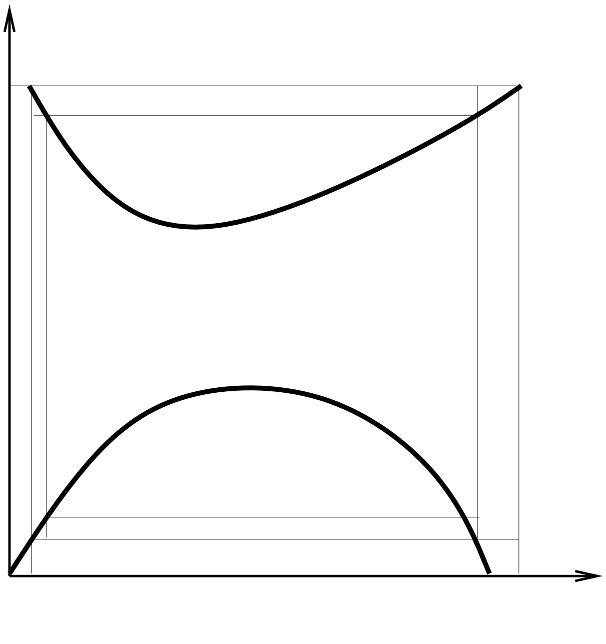

Figure 15: Case of slow convergence of domain decomposition

One such example is described in fig.15. 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). Domain decomposition is then performed on each half. The step size of tracing is determined by the size of each region.

4.2 Tracing in lower dimension

After domain decomposition, the entire domain ($[0,1] \times [0,1]$) is subdivided into smaller regions each with exactly one curve segment. Further, domain decomposition 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.

- 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 normal to the plane curve. Methods to compute the partial derivatives was described in section 2 on matrix computations.
- 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.
- 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.9(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 $\mathbf{Q_3} = (u_2, v_3)$.

Power Iterations: Power iterations are a fundamental technique to compute the eigenvalues and eigenvectors of a matrix. Given a diagonalizable matrix, $\mathbf{A}$, we can find $\mathbf{X}(= [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n])$ such that $\mathbf{X}^{-1}\mathbf{A}\mathbf{X} = Diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$. Given a unit vector $\mathbf{q}_0$, the *power method* produces a sequence of vector $\mathbf{q}_k$ as follows:

$$\mathbf{z}_k = \mathbf{A}\mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{A}\mathbf{q}_k;$$

where $\parallel \mathbf{z}_k \parallel_\infty$ refers to the infinity norm of the vector $\mathbf{z}_k$. s_k converges to the largest eigenvalue λ_1 and $\mathbf{q}_k$ converges to the corresponding eigenvector $\mathbf{x}_1$.

In the tracing algorithm, we compute the eigenvalue of $\mathbf{M}(u_2, v)$ which is closest to v_2 (fig.9(d)). As a result, we compute the companion matrix $\mathbf{C}$ from $\mathbf{M}(u_2, v)$ (see eq. (3)) and set $s = v_2$. In our application, 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. Given an initial unit vector $\mathbf{q}_0$, we generate a sequence of vectors $\mathbf{q}_k$ as

$$\textit{Solve}\ (\mathbf{C}-s\mathbf{I})\mathbf{z}_k = \mathbf{q}_{k-1}; \quad \mathbf{q}_k = \mathbf{z}_k / \parallel \mathbf{z}_k \parallel_\infty; \quad s_k = \mathbf{q}_k^T \mathbf{C} \mathbf{q}_k;$$

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 the complexity of LU decomposition. Given s, let $\mathbf{B}=\mathbf{C}-s\mathbf{I}$. $\mathbf{B}$ is of the form:

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

where α_1 is a function of s, and $\mathbf{P}_i$'s are $n \times n$ matrices which are functions of $\mathbf{M}_i$'s and s. The LU decomposition of $\mathbf{B}$ has the form:

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

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. The order of $\mathbf{B}$ is $2mnd$. However the LU decomposition takes only $\frac{1}{3}d(2mn)^3$ operations. This structure is utilized in solving the lower and upper triangular systems at each step as well.

A key property of inverse power iteration is that it converges to the eigenvalue closest to s. The initial vector $\mathbf{q}_0$ is chosen randomly. In case the closest eigenvalue to s is a complex conjugate pair, the power method involving real operation isolates and does not converge to any real value. In such cases, the tracing algorithm chooses a smaller step size for computation. We refine the solution obtained from inverse power iterations using minimization methods on the distance function $D(u,v,s,t)$ defined in eq. (7). Inverse power iterations followed by a few minimization steps give very good accuracy in practice.

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.9(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 S. (This statement is not true in regions of very high curvature or cusps. These cases can be handled separately.) If $\delta < \Delta$, the inverse power iteration guarantees that after the convergence of power iteration the resulting point is still on $\mathbf{C}$. Therefore, an upper 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 domain decomposition and inverse power iterations, we ensure that there is no component jumping during tracing.

In the next section, detection of cusps during tracing will be discussed. As we will see, cusps in the plane curve need not necessarily correspond to a singular point on the intersection curve (in space), and are therefore much more difficult to handle accurately.

5 Singularities

In this section, we describe algorithms to detect cusps. Cusps are points on the curve that cannot be detected by the domain decomposition algorithm readily. Therefore some more work has to be done to detect the presence of cusps. The tracing algorithm evaluates an algebraic plane curve ($D(u, v) = 0$). Singularities on the plane curve $D(u, v) = 0$ are characterized by the common solutions of $D(u, v) = D^u(u, v) = D^v(u, v) = 0$. The different kinds of singularities are shown in figs.$16(a), 16(b), 16(c)$. Algebraically the singularities are classified by the number of branches or *places* the curve has at that point [AB88]. Cusps have only one branch while nodes and loops have more than one branch.

Singularities on the intersection curve correspond to points where the tangent vector is undefined. The tangent to the intersection curve is obtained by taking the cross-product of the surface normals at that point. As a result, the preimages of singular points on the intersection curve $\mathbf{I}$ are the common solutions of

$$\begin{aligned} \mathbf{F}(s, t) &= \mathbf{G}(u, v) \\ (\mathbf{F}^s(s, t) \times \mathbf{F}^t(s, t)) \times (\mathbf{G}^u(u, v) \times \mathbf{G}^v(u, v)) &= (0 \quad 0 \quad 0)^T \end{aligned} \tag{9}$$

The curve $\mathbf{I}$ may have more than one branch at the singularity. The *nodes* on the intersection curve and the plane curve are related by the following lemma.

Lemma 2 *If the surface has no self-intersections, a* node *on the plane curve $D(u, v) = 0$ corresponds to a* node *on the intersection curve* $\mathbf{I}$.

Proof: A patch is said to be *faithfully* parametrized if the mapping from parametric space to the surface is bijective. In other words, there are no self-intersections on the surface.

Let $\mathbf{Q}$ be a singular point of the curve in the domain and $\mathbf{P}$ be its image on $\mathbf{I}$. Corresponding to each branch of $\mathbf{P}$, there is a sequence of points on the curve $D(u, v) = 0$ converging to $\mathbf{Q}$. The images of these points on the intersection curve converge onto $\mathbf{P}$. The one-to-one mapping would imply that there are different branches in the neighborhood of $\mathbf{P}$ as well. Therefore, $\mathbf{P}$ corresponds to a *node*.

$\square$

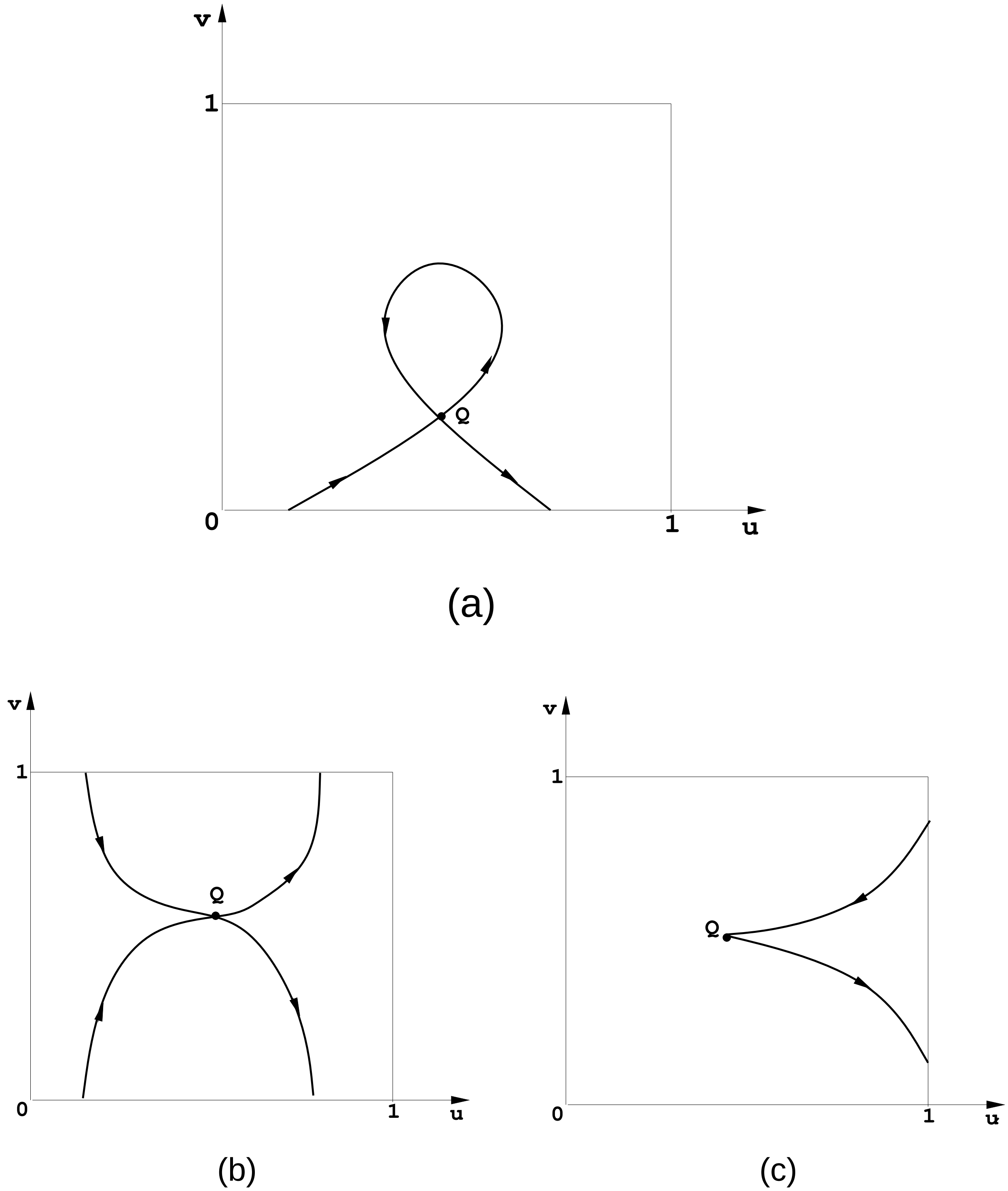

Figure 16: Types of singularity (a) Loop (b) Tacnode (c) Cusp

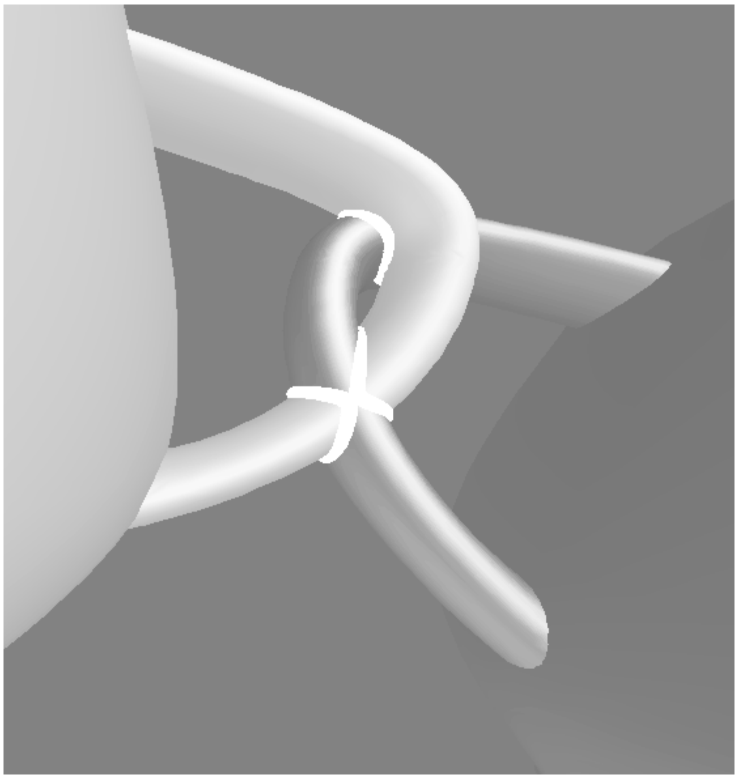

Figure 17: Teapot handles intersecting at a tacnode

However, there is no direct relationship between cusps on the plane curve and those on the intersection curve. In particular, $D(u, v) = 0$ may have cusps but $\mathbf{I}$ need not have cusps and vice-versa. Since we are evaluating the plane curve, we compute all the singularities and the branches.

In general the problem of computing the singularities in the intersection curve of high degree surfaces in floating point arithmetic can be numerically unstable [FR87]. Algorithms based on exact arithmetic and birational transformations have been proposed in [AB88]. However they are computationally very slow. Our algorithms are based on the local geometry of the curve and the properties of the representation $\mathbf{M}(u, v)$. The algorithm proposed here assumes that all the singular points on the curve are geometrically isolated and well apart.

5.1 Detection of Cusps

The previous section described a method of detecting nodal singularities during domain decomposition. Domain decomposition made use of the fact that any region with just two boundary intersections consists of a single curve component. However this curve could contain cusps which introduce problems during tracing. Cusps on the plane curve are computed based on the following lemma.

Lemma 3 *Given a singular point* (u_1, v_1) *on the curve, one of the following must be true,*

- $\mathbf{M}(u_1, v_1)$ *has more than one zero singular value, or*
- *the entries* $g^u_{nn}(u_1, v_1)$ *and* $g^v_{nn}(u_1, v_1)$ *obtained after performing Gaussian elimination of* $\mathbf{M}(u_1, v_1)$ *are both zero.*

Proof: In section 2 it was shown that using a slight variant of Gaussian elimination, $D(u_1, v_1) = \Pi_{i=0}^{n} g_{ii}$ and $D^u(u_1, v_1) = D(u_1, v_1) \sum_{i=0}^{n} \frac{g^u_{ii}}{g_{ii}}$. It is a well known fact that if complete pivoting is used during Gaussian elimination then for $i < j$, $g_{ii} \geq g_{jj}$.

Since (u_1, v_1) is a point on the curve, $D(u_1, v_1) = 0$. Therefore, at least $g_{nn}(u_1, v_1) = 0$. In addition, since (u_1, v_1) is also a singular point, $D^u(u_1, v_1) = D^v(u_1, v_1) = 0$. Thus,

$$\Pi_{i=0}^{n} g_{ii}(u_1, v_1) \quad = \quad \Pi_{i=0}^{n} g_{ii}(u_1, v_1) \sum_{i=0}^{n} \frac{g^u_{ii}(u_1, v_1)}{g_{ii}(u_1, v_1)} \quad = \quad 0.$$

This implies one of two cases -

- at least another $g_{ii}(u_1, v_1) = 0, i \neq n$, or
- $g^u_{nn}(u_1, v_1) = g^v_{nn}(u_1, v_1) = 0$.

The latter case satisfies the second part of the lemma. If the former is true, then $\mathbf{M}(u_1, v_1)$ must be rank deficient by at least two. This is equivalent to two or more singular values being zero.

$\square$

The tracing algorithm computes the partial derivatives at each stage and checks for the conditions in Lemma 3. In case one of them is *approximately* satisfied, we conjecture that we are near a cusp. Tracing is then abandoned on this path and started from the other endpoint. If the curve has a cusp in this region, the paths from the two endpoints meet at this cusp. Once the two paths come close (as demanded by the application) to each other the tracing stepsize is progressively reduced. Once the two paths are close enough (smaller than specified tolerance), the cusp point is obtained by minimizing an energy function. The energy function is determined by the condition satisfied in Lemma 3. For example, if the second condition is satisfied, the energy function is $E(u, v) = (g^u_{nn}(u, v))^2 + (g^v_{nn}(u, v))^2$.

6 Robustness and Efficiency

The two most important considerations in the design of any surface intersection algorithm are *robustness* and *efficiency*. There is a clear trade-off between these two, since the larger the robustness enhancement computations the slower the execution time of the algorithm. While it is almost impossible to provide an algorithm that can satisfy both completely together, one must at least target towards 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.

Our algorithm is designed to do just that. We have tested our algorithm on a variety of models, and have obtained encouraging results. There are no benchmarks available to test its efficiency, but our algorithm compares favorably to many of the published timings. For example, it performs an order of magnitude faster than techniques like *interval arithmetic*. It is also possible to for us to perform extra work in order to give more robust results.

In order to guarantee robustness, a general intersection algorithm must be able to determine the *conditioning* of the problem. 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 surface intersection problems. Because of such difficulties, we restrict ourselves to robustness issues for well-conditioned problems only.

We identify four main areas in our algorithm where robustness enhancing modifications can be made. They are

- **tracing -** We perform tracing after domain decomposition. It is shown that using inverse power iterations convergence back to the curve is guaranteed except in places of very high curvature (or cusps). In such situations, we reduce the step size by half and repeat the process. Robustness can be enhanced by reducing the *minimum step size*.

- **component jumping -** Domain decomposition is adopted to prevent component jumping. Robustness is determined by the extent to which subdivision is carried out. It is possible to control that effectively based on the requirement of the application.

- **singularities -** The accurate detection of all the singular points is contingent on the assumption that the singular points are separated and well apart. This assumption is not unreasonable for most of the practical applications, but a pathological case violating this condition can be created.

- **loop identification -** We identify loops by performing curve-surface intersection and complex tracing. The number of paths to be traced grows as a cubic function of the degree of a patch. In most practical examples however, it is usually enough to trace a small fraction of the paths. But to ensure absolute robustness, all the paths have to be traced.

7 Large Scale Models

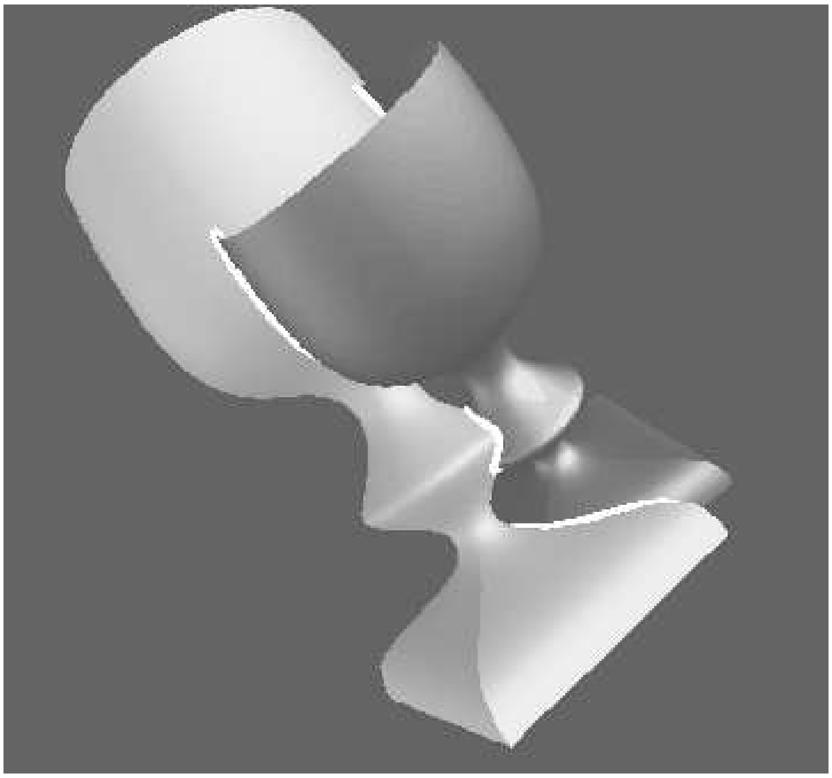

Figure 18: Intersecting Goblets

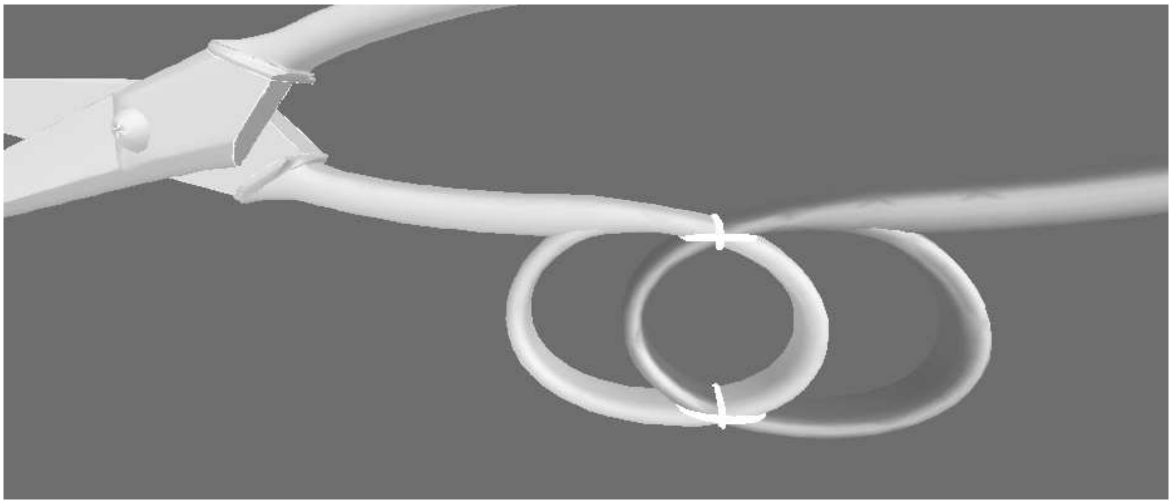

Figure 19: Intersecting Scissors

Model	No. of patches	No. of intersections	Timing (in sec.)		
			Curve-Surf. int.	Tracing	Total
Teapot	32	8	0.4	2.6	3.5
Goblet	72	57	2.4	4.9	7.8
Scissor	505	82	3.1	7.8	11.6

Table 1: Performance Statistics of Intersection Algorithm

The algorithms in sections 3, 4 and 5 deal with the intersection of a pair of Bézier or algebraic surfaces only. Most of the models consist of tens or hundreds of such surface patches. To compute the intersections of these models we compute the intersection of each overlapping pair.

Typically only a small percentage of the $O(N^2)$ possible surface pairs intersect. Our algorithm prunes out most of non-intersecting combinations using spatial techniques based on bounding boxes and linear programming.

- Initially the axis-aligned rectangular bounding boxes are computed for each Bézier surface for both models. Each of these bounding boxes are then projected on each of the three axes to obtain three sets of intervals. We denote them by *x-lists, y-lists and z-lists*. The *x-lists* is sorted first and all the non-intersecting pairs are discarded. The *y-lists* of the remaining pairs are again sorted to check for overlaps. This is repeated for the *z-lists* as well. The pairs that remain at the end of this operation have intersecting bounding boxes.

- The Bézier patches are contained in the convex hull of their control points. Given all the pairs of surfaces obtained after bounding box tests, a test for separating plane between their control polytopes is performed using linear programming. The existence of a separating plane implies that the surface pairs have no intersection.

After execution of these two steps, the intersection algorithm is applied to each existing pair. The intersection of each patch pair results in a set of open components and loops. The open components could be part of a larger curve in the model. Two open components are spliced together if one of each of their endpoints are coincident (actually the test is made over a small disk of influence). Finally a piecewise representation of each component of the intersection curve is obtained for the original models. The results of the intersection algorithm on large models like goblets and scissors are shown in figs. 17 and 18.

8 Implementation and Performance

The algorithm has been implemented and its performance was measured on a number of models. The algorithm uses existing EISPACK and LAPACK 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 run on a high-end SGI Onyx workstation with a peak floating-point performance of 300 MFLOPS. In Table 1, we have highlighted the algorithm's performance on different models. The second column indicates the number of patches in each model, and the third column corresponds to the number of patch pair intersections after linear programming. 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.

9 Future Work

Surface intersection for general parametric surfaces is a hard problem. A number of issues are still open regarding the numerical aspects and efficiency of practical surface intersection algorithms. In this paper, we identify three main problems still plaguing practical systems. They are

- **Good tolerance adjustment -** Geometric algorithms implemented on fixed precision arithmetic introduce different types of computational errors. Therefore different comparison tests have to be made using tolerances. Due to the wide range of input values encountered, setting a fixed tolerance is not sufficient. Currently our system normalizes the patches (scaled down) before applying the intersection algorithm, but still the tolerance adjustments are not adequate.

- **Estimation of conditioning -** The conditioning of a problem measures the stability of the results (by applying some algorithm) on small perturbations of the input data. In order to guarantee any form of robustness, an algorithm must be able to compute the *condition number*. One obvious estimate of conditioning is the angle at which the two surfaces intersect. But that is not easily computable.

- **Robust and efficient loop detection -** Our loop detection algorithm can be made robust by tracing all the complex paths. But rapid increase in complexity (with degree) makes it somewhat infeasible for high-degree intersections. There does not seem to be any loop detection algorithm that can ensure correctness in a reasonable amount of time.

10 Conclusion

We have presented an efficient algorithm for evaluating the surface intersection of parametric surface models. The algorithm has been implemented in floating points arithmetic and performs well in practice. Its main characteristics are computation of start points, loop detection based on complex tracing, robust tracing without component jumping and evaluation of all singular points. At the same time its overall performance on large scale models is fast enough for current CAD systems.

11 Acknowledgement

We are grateful to E. Cohen, R. Fish and R.Riesenfeld for the goblet and scissor models. These models were created using the *Alpha-1* CAD system.

References

[AB88] S.S. Abhyankar and C. Bajaj. Automatic parametrizations of rational curves and surfaces III: Algebraic plane curves. *Computer Aided Geometric Design*, 5:309–321, 1988.

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

[BK90] R.E. Barnhill and S.N. Kersey. A marching method for parametric surface/surface intersection. *Computer Aided Geometric Design*, 7:257–280, 1990.

[Che89] K.P Cheng. Using plane vector fields to obtain all the intersection curves of two general surfaces. In *Theory and Practice of Geometric Modeling*, pages 187–204, 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.

[Far86] R.T. Farouki. The characterization of parametric surface sections. *Computer Vision, Graphics and Image Processing*, 33:209–236, 1986.

[FF92] D.A. Field and R. Field. A new family of curves for industrial applications. Technical report GMR-7571, GM Labs, 1992.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[Gei83] A. Geisow. *Surface Interrogations*. PhD thesis, School of Computing Studies and Accountancy, University of East Anglia, 1983.

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

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

[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.

[KPP90] G.A. Kriezis, P.V. Prakash, and N.M. Patrikalakis. Method for intersecting algebraic surfaces with rational polynomial patches. *Computer-Aided Design*, 22(10):645–654, 1990.

[KPW90] G.A. Kriezis, N.M. Patrikalakis, and F.E. Wolter. Topological and differential equation methods for surface intersections. *Computer-Aided Design*, 24(1):41–55, 1990.

[Kriez90] G.A. Kriezis. Algorithms for rational spline surface intersections. *Ph.D. Thesis*, March 1990.

[Las86] D. Lasser. Intersection of parametric surfaces in the bernstein-bezier representation. *Computer-Aided Design*, 18(4):186–192, 1986.

[LR80] J.M. Lane and R.F. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159, 1980.

[M93] D. Manocha. Algebraic Pruning: A fast technique for curve and surface intersections. *Tech. Report: Univ. of North Carolina, Chapel Hill*, TR93-062, 1993.

[Man93] D. Manocha. Solving polynomial systems for curve, surface and solid modeling. In *ACM/SIGGRAPH Symposium on Solid Modeling*, pages 169–178, 1993.

[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.

[Pat93] N.M. Patrikalakis. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95, 1993.

[Piegl89] L. Piegl. Geometric methods of intersecting natural quadrics represented in trimmed surface form. *Computer Aided Design*, 21(4):201–212, May, 1989.

[Pra86] M.J. Pratt. Surface/surface intersection problems. In J.A. Gregory, editor, *The Mathematics of Surfaces II*, Oxford, 1986. Claredon Press.

[RMS92] J. Rossignac, A. Megahed, and B.D. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[RR87] J.R. Rossignac and A.A.G. Requicha. Piecewise-circular curves for geometric modeling. *IBM Journal of Research and Development*, 31(3):296–313, 1987.

[RR92] A.A.G. Requicha and J.R. Rossignac. Solid modeling and beyond. *IEEE Computer Graphics and Applications*, pages 31–44, September 1992.

[Sar83] R F Sarraga. Algebraic methods for intersection. *Computer Vision, Graphics and Image Processing*, 22:222–238, 1983.

[Sed83] T.W. Sederberg. *Implicit and Parametric curves and surfaces.* PhD thesis, Purdue University, 1983.

[SM88] T.W. Sederberg and R.J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5:161–171, 1988.

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

[Sei90] Raimund Seidel. Linear programming and convex hulls made easy. *ACM symposium on computational geometry*, 211–215, ACM Press, 1990.

[SM94] S. Krishnan and D. Manocha. Intersection computations on parametrizations with base points. *under preparation*, 1994.

[SKW85] Pradeep Sinha, Eric Klassen, and K. K. Wang. Exploiting topological and geometric properties for selective subdivision. *ACM symposium on computational geometry*, 39–45, ACM Press, 1985.

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

[ZS93] A. Zundel and T. Sederberg. Using pyramidal surfaces to detect and isolate surface/surface intersections. In *SIAM Conference on Geometric Design*, Tempe, AZ, 1993.