Efficient and Accurate B-rep Generation of Low Degree Sculptured Solids Using Exact Arithmetic:II - Computation

John Keyser [a], Shankar Krishnan [b], and Dinesh Manocha [a]

[a] *Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599, USA*

[b] *AT&T Research Labs, Florham Park, NJ 07932, USA*

Abstract

We present efficient algorithms for exact boundary computation on low degree sculptured CSG solids using exact arithmetic. These include algorithms for computing the intersection curves of low-degree trimmed parametric surfaces, decomposing them into multiple components for efficient point location queries inside the trimmed regions, and computing the boundary of the resulting solid using topological information and component classification tests. We also employ a number of previously developed algorithms such as algebraic curve classification and multivariate Sturm sequences. We present some results from a preliminary implementation of our approach. This paper follows a previous paper which described the representations used in our approach.

Key words: Solid modeling, exact arithmetic, robustness, boundary computation

1 Introduction

The computation of boolean combinations of solids is an important operation in several modeling systems. It is a key component in several approaches for converting between from the CSG (constructive solid geometry) representation

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

to a B-rep (boundary representation). Performing such operations accurately and robustly, particularly on curved solids, has proven to be difficult in practice. Major reasons for this include the inaccuracies inherent to floating-point (or other non-exact) arithmetic and dealing with degeneracies.

We have developed an approach for finding boolean combinations of low-degree sculptured solids using exact arithmetic. This paper is a followup to a paper [8] which discussed, among other things, the representations used in our approach. We assume that the reader is familiar with the previous paper, but we briefly review some key points regarding the representation.

We represent solids as a number of trimmed patches, maintaining the topological structure of the patches in each solid. Each patch is part of a parametric surface, where the rational parameterization defines X,Y,Z, and W as polynomial functions of the patch parameters, s and t, with rational number coefficients. The trimming curves are algebraic plane curves, again expressed as polynomials with rational coefficients. Intersections of these trimming curves (i.e. the vertices) are defined using rational rectangles guaranteed to isolate a single intersection of the two curves.

Main Contribution: We present an efficient approach for evaluating Boolean combinations of low-degree sculptured solids using exact arithmetic. Our approach is efficient in the sense that it is significantly faster than other proposed approaches based on exact computation for algebraic primitives, which use worst-case bounds on bit length or use general purpose algebraic solvers (as available in computer algebra systems). The approach presented does not completely address the robustness issue, since it does not effectively deal with all degeneracies, although elimination of numerical errors alleviates some of the robustness problems. Our contributions in this paper include:

- **Kernel Routines:** We identify lower-level routines where the algorithms based on floating-point arithmetic are susceptible to failure. These include sign-evaluation of geometric predicates, orientation of points with respect to curves, and component classification. We present fast algorithms to perform such tests *reliably* using exact arithmetic on our exact representations. We refer to the resulting set of routines as *kernel routines*. The efficiency and reliability of the overall algorithm is governed by these routines.
- **B-rep Computation:** We present details of the steps in our algorithm for computing the boolean combinations. We provide details for how this is done, given the kernel routines described, while maintaining an exact
- **Implementation and Performance:** We describe the performance of a preliminary implementation of our algorithms.

Organization: The rest of this paper is organized as follows. Section 2 discusses our kernel routines. Section 3 gives an in depth description of the steps

in the algorithm. Section 4 discusses analysis and performance details. Section 5 provides a small example of how parts of the algorithm work.

A preliminary version of this paper was presented at the Solid Modeling '97 conference [9].

2 Exact computation of kernel routines

In this section, we present efficient algorithms to implement the kernel routines in exact arithmetic. The kernel routines, as well as where they are used, are identified in Fig. 1. The primary kernel routines—multipolynomial resultants and multivariate Sturm sequences, were discussed in the appendix of a previous paper [8], but we review multivariate Sturm sequences briefly here. In particular, given algebraic curves and points, we present here efficient algorithms for comparing two algebraic numbers, evaluating signs of slopes for resolution of regular algebraic curves, and classifying points with respect to a region.

2.1 Multivariate Sturm Sequences

Multivariate Sturm sequences form a key part of our approach, since we use them to determine the boxes which form the vertices of our solid. We use multivariate Sturm sequences, as described by Milne [13], to find the number of intersections between two algebraic plane curves within a rectangular region. We omit most details here, but provide a brief overview of major steps in building a Sturm sequence. For more details, see [13] or the appendix to our previous paper [8].

Given two polynomials, $f_1(s,t)$ and $f_2(s,t)$, we construct the *volume function*, $V(u,s,t)$, as follows:

$$V(u,s,t) = \frac{Res_{a_2}(Res_{a_1}(f_1(a_1,a_2),f_3),Res_{a_1}(f_2(a_1,a_2),f_3))}{u^{deg(f_1(s,0))deg(f_2(s,0))}},$$

where $f_3(u,s,t,a_1,a_2) = u + (s-a_1)(t-a_2)$, Res_x refers to the resultant of two polynomials after eliminating x, and deg refers to the degree of the polynomial.

Treating V as a univariate polynomial in u, we construct its Sturm sequence. The number of sign changes in this Sturm sequence when $u = 0$ is a function in terms of s and t. By plugging in the coordinates at the corners of a rational

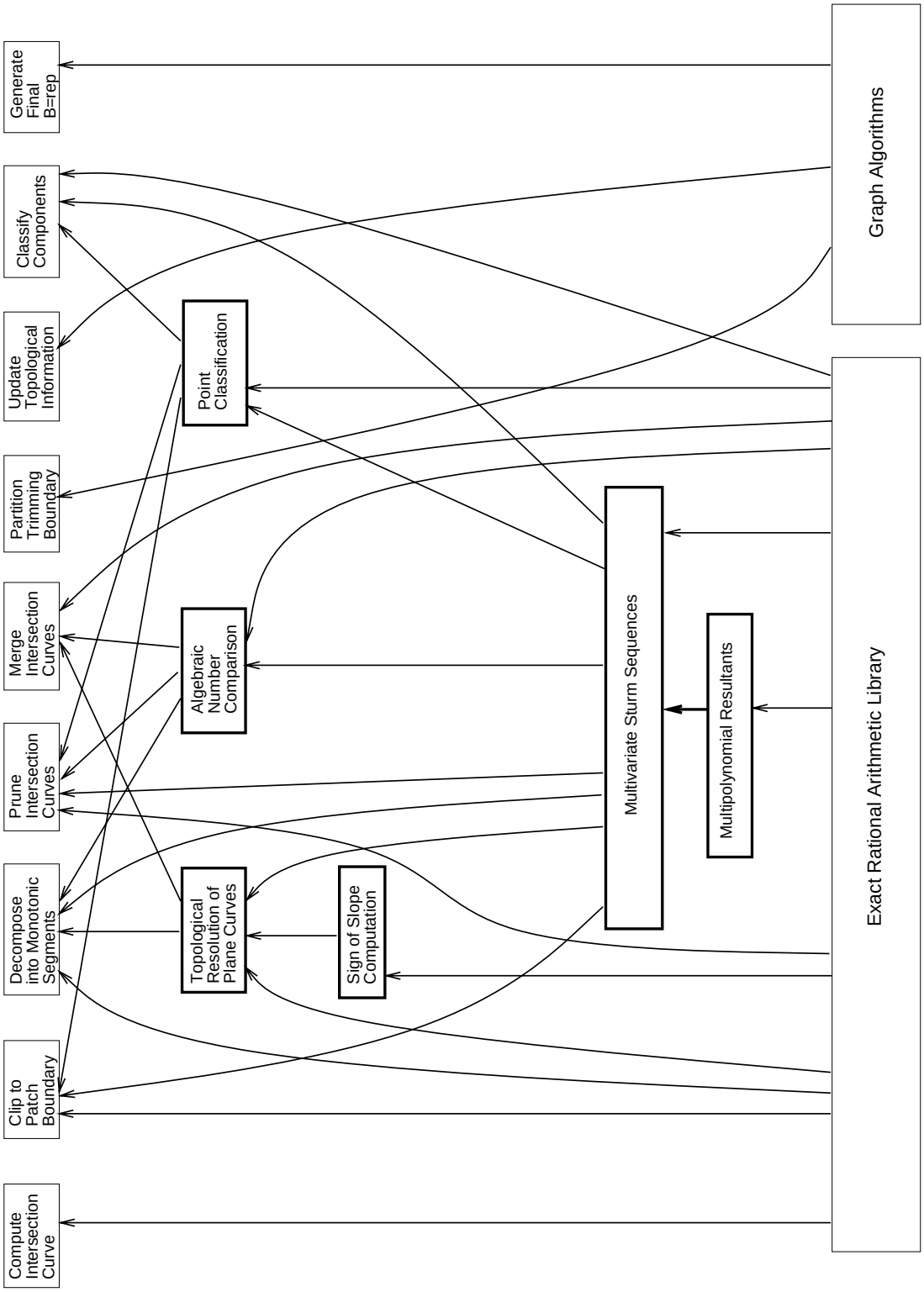


Fig. 1. An overview of the algorithm's components. The kernel routines are those outlined with bold boxes. Arrows indicate dependencies between various components.

rectangle for s and t, the number of intersections between $f_1 = 0$ and $f_2 = 0$, within that rectangular region, is easily computed.

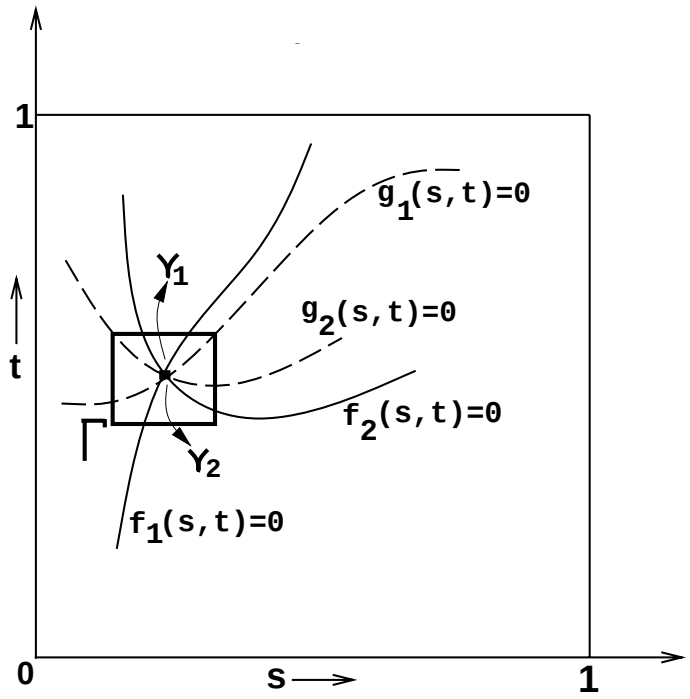
2.2 Comparison between algebraic numbers

We have mentioned that the vertices in the solid are defined as roots of a set of polynomial equations with rational coefficients. Since we are dealing with rational parametric surfaces, each of these equations is bivariate. A vertex in the patch domain is the common solution of two equations, $f_1(s,t) = 0$ and $f_2(s,t) = 0$. This is usually an algebraic number which cannot be represented exactly using floating-point arithmetic. In our algorithm, we represent each real algebraic number using a small rational rectangle. The rational rectangle is guaranteed to isolate each common root of $f_1(s,t)$ and $f_2(s,t)$ (taking into account the multiplicities of roots).

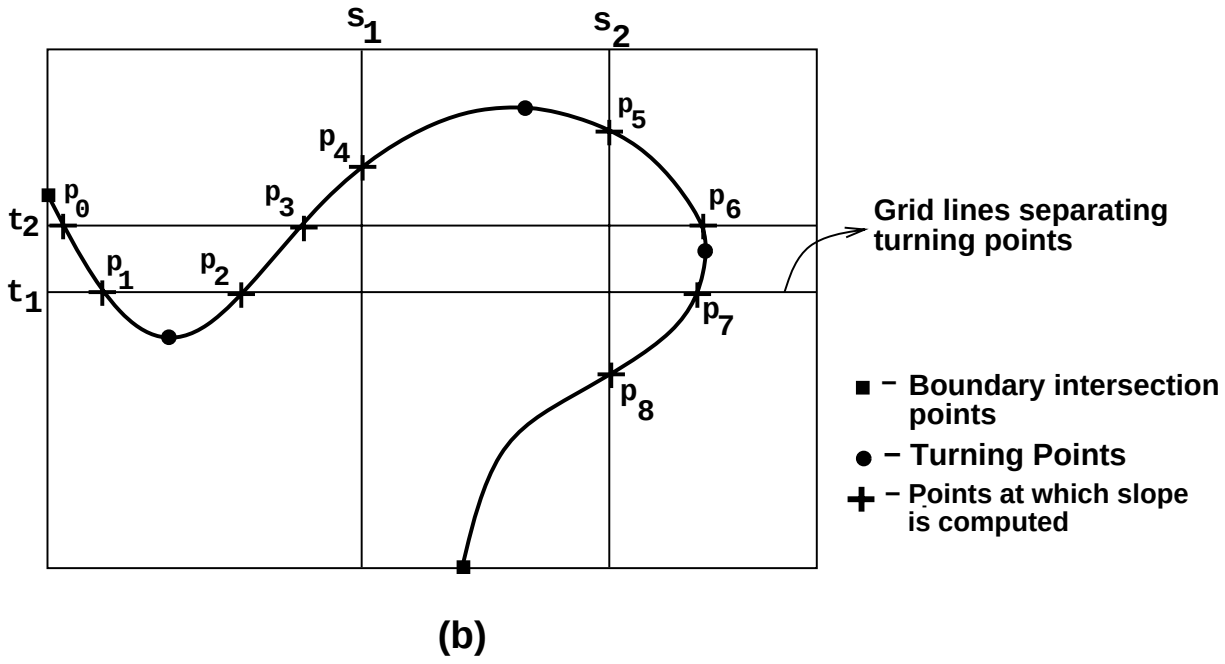
Consider a rational rectangle, , , that contains two algebraic numbers, γ_1 and γ_2 (see Fig. 2(a)). Let γ_1 be the common root of $f_1(s,t)$ and $f_2(s,t)$, and γ_2 the common root of $g_1(s,t)$ and $g_2(s,t)$. There is an exact procedure to answer the question of equality of γ_1 and γ_2. Consider the new polynomial $p(s,t) = f_1^2(s,t) + f_2^2(s,t) + g_1^2(s,t) + g_2^2(s,t)$, and one of the original polynomials (say $f_1(s,t)$). It is easy to show that $p(s,t)$ and $f_1(s,t)$ have a common isolated root in , iff $\gamma_1 = \gamma_2$. We check for the common root using Sturm sequences. Further, this method is extensible to arbitrary dimensions. This computation can be very expensive. Because of this, it is a good idea to narrow the bounding rectangles to a small interval first. Only after the rectangles are both very small should the exact equality test be performed.

2.3 Topological resolution of algebraic plane curves

The intersection curve between two surfaces is typically a high degree algebraic curve. In practice, it may have multiple real components. Topological resolution involves identifying critical points like turning points and singularities (self intersections and points where the tangent vanishes) and establishing a unique connectivity between them. General algorithms to solve this problem (such as Cylindrical Algebraic Decomposition) can have doubly exponential running times, whcih can take a considerable amount of time even on small cases, in practice. For special kinds of algebraic curves, a number of efficient (poly-log time) algorithms have been developed. We use the algorithm by [1] for regular curves. The algorithm initially computes all the turning points of the curve. This is achieved in our case by taking partial derivatives and solving for common roots with the original curve inside a rational rectangle. A crucial step in establishing connectivity between the various turning points is to find



(a)



(b)

Fig. 2. (a) Algebraic number comparison (b) Topological resolution of algebraic curves in finite domain

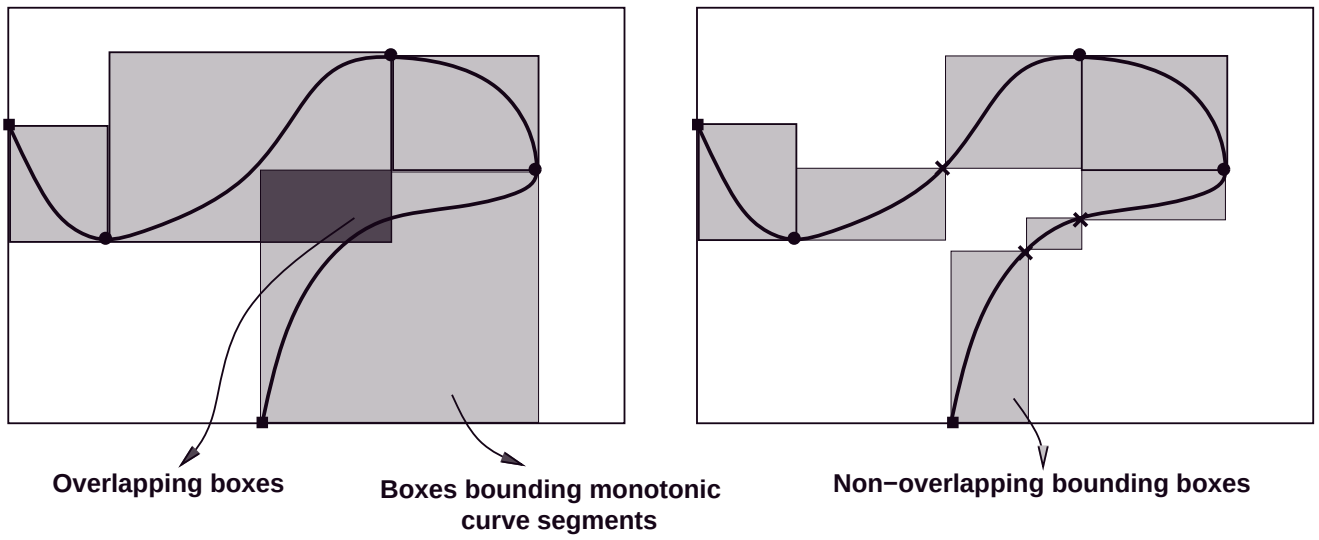


Fig. 3. Bounding boxes around monotonic curve segments

the sign of the slope of the curve at certain irrational (algebraic) points. We present an exact algorithm to perform this step next.

2.3.1 Slope sign computation at algebraic points

Fig. 2(b) shows an example of an algebraic curve in a small region. Once the turning points are computed, they are separated using rational grid lines (s_1, s_2, t_1, and t_2 in the figure). In order to determine the topology of the curve, we need to compute the sign of the slope of the curve at points where the grid lines intersect the curve (e.g. p_4 in figure). Since p_4 is algebraic (a root of a univariate polynomial), we can compute it only within an interval (say, $[t_{41}, t_{42}]$). It is non-trivial to compute the sign of the slope directly (equivalent to computing signs of partial derivatives) at p_4. Let $f(s,t)$ be the equation of the curve. We know that there is a unique root of $f(s_1,t)$ in the interval $[t_{41}, t_{42}]$. Let $g(t) = f_s(s_1,t)$. We know that p_4 is not a common root of $f(s_1,t)$ and $g(t)$ (because it is not a turning point). We refine the interval $[t_{41}, t_{42}]$, "zooming in" on p_4, until there is no root of $g(t)$ within the interval. The sign of $g(t)$ can then be obtained by evaluating $g(t)$ at any rational point in the interval. The sign of $f_t(s_1,t)$ is found in an analogous manner. The sign of the slope, then, is found using the relation $dt/ds = -f_s/f_t$.

2.3.2 Finding bounding boxes

The topological resolution of the intersection curve divides it into a set of curve segments that are monotonic in s and t, the patch parameters. For each of these segments, we compute a bounding box (see Fig. 3) around it. Bounding boxes are used to distinguish two such curve segments represented by the same algebraic equation. However, as seen from Fig. 3, not all the bounding boxes are non-overlapping. We perform a subdivision of these boxes until they no longer overlap. We use an algorithm described in [11] to perform this subdivision. The bounding boxes defined here are used to classify a point efficiently with respect to a curve.

2.4 Point classification

Classifying a component with respect to a solid amounts to classifying a point with respect to the trimmed domain. Problems associated with point classification using floating-point arithmetic were highlighted earlier. We now describe our algorithm to exactly check whether a point (with algebraic number coordinates) lies inside or outside the trimming boundary. Initially, we assume that the actual point does not lie exactly on any algebraic curve that is part of the trimming boundary. This would be a case of four surfaces intersecting at a point and will be discussed when we address degeneracies.

Given that the point is an algebraic number, we represent it with a rational rectangle of small size. We must ensure that the rectangle lies in at most one of the bounding boxes of the trimming curve (monotonic segments and bounding boxes are discussed in the appendix, section 8.3). If it lies in more than one box, the rectangle should be further refined so that it lies in only one. Each of the four corners of the rectangle is then classified with respect to the trimming boundary. This classification is done using a ray shooting technique. To determine if a point lies inside or outside the trimming boundary, we shoot an arbitrary semi-infinite ray from it. To do this, just choose any line through the rational point (horizontal or vertical lines make the computation simpler). Then, compute all intersections (to one side of the point) of the line with the boundary. The parity of the number of intersections of the ray with the boundary is sufficient to classify the point.

We perform this test on each of the four corners of the rectangle. If all the results are the same, the point is classified. The more interesting case is when some corners of the rectangle yield different results. In this case, we refine our rectangle until consistency is achieved. Eventual consistency is assured because the point does not lie on any boundary curve. If the 2-D point arises from a 3-D ray intersection (during component classification), we can always

choose a different 3-D ray if 2-D classification takes more than a few levels of refinement.

3 B-rep computation algorithm

In this section, we elaborate on each step of our algorithm for determining Boolean combinations of sculptured solids. It is implemented on top of the kernel routines. Each of the following subsections corresponds to a step in the algorithm outline, as shown in Fig. 1.

3.1 Compute intersection curve for two patches

Our goal is to compute the intersection curve in the domain of each of the two patches. To find the intersection curve in the domain of patch 1, we substitute the parameterization of patch 1 into the (precomputed) implicit representation of patch 2. A similar procedure obtains the intersection curve in the domain of patch 2. If we are not given an implicit representation, we implicitize the parametric surface as described in the appendix (section 8.1).

3.2 Clip to patch boundary

Curve-surface intersection is used to find the points where the intersection curve meets the patch boundary. We want to find the intersection points of the curve and surface in the domain of both patches. Finding the intersection points of a curve with the boundaries of its own patch is straightforward. The patch boundaries (not the trimmed boundaries) are values of s and t, for example $s = 0$. Substituting the value for s (or t) into the equation of the intersection curve gives a univariate equation. The roots of this univariate equation (found using univariate Sturm sequences) then give the values for t (or s) of the intersection points along that boundary. The *inversion* problem is to find the corresponding points in the domain of patch 2.

One way to solve the inversion problem is to consider the patch boundary as a space curve in $\Re^3$. We compute the intersections of this space curve with the other patch (a curve-surface intersection). For example, consider the domain boundary $t = 0$. Then, the space curve corresponding to the patch boundary will be given by $X(s,0), Y(s,0), Z(s,0)$, and $W(s,0)$, where X, Y, Z, W give the parameterization of the first patch. The second patch will have the parameterization $\overline{X}(u,v), \overline{Y}(u,v), \overline{Z}(u,v), \overline{W}(u,v)$. Then, to solve for the points

of intersection we need to solve for roots of the equations (1) in the domain of each patch.

$$\begin{aligned} \overline{X}(u,v)\ W(s,0)\ &-\ X(s,0)\ \overline{W}(u,v) = 0 \\ \overline{Y}(u,v)\ W(s,0)\ &-\ Y(s,0)\ \overline{W}(u,v) = 0 \\ \overline{Z}(u,v)\ W(s,0)\ &-\ Z(s,0)\ \overline{W}(u,v) = 0 \end{aligned} \tag{1}$$

These equations may be solved using a trivariate Sturm sequence as described in [13]. This will give solutions bounded in u, v, and s. However, the volume function computation involves three successive elimination steps (Sylvester resultant). Depending on the size of the coefficients, this process can be computationally intensive. Multivariate Sturm sequences and resultants are discussed in the appendix.

A more practical, though not exact, approach to solve this problem is to isolate all the roots in the s domain using a univariate Sturm sequence. This is followed by eliminating s from (1) to produce two independent equations in u and v. This is a bivariate Sturm sequence problem and is solved by a single Sylvester resultant computation. This gives all the solutions in (u, v) space. Determining the correspondence between the (u, v) pairs and s roots is done by comparing each of them in 3-space. We found that in practice, this method was significantly more efficient. This approach is inexact since we do not usually compute all solutions in (u, v) space, but rather only the solutions within some interval (the patch boundary). If we can guarantee that *all* real solutions in both the s and the (u, v) domains were found, then this approach is also exact (assuming we use an exact 3-space matching algorithm).

3.3 Decompose into monotonic curve segments

The intersection curve is now divided up into segments which are monotonic in s and t. A process for doing this is discussed in the appendix (section 8.3). Significant parts of this step are finding the turning points (done using multivariate Sturm sequences, also discussed in the appendix), and computing the sign of the slope of a curve, which is discussed in section 3.3.2.

3.4 Prune intersection curves

We now have to trim the curve based on the trimming boundary. Basically, we need to intersect the intersection curves with the trimming curves and

clip away sections of the intersection curve which are outside the trimming boundary.

Finding the points of intersection between the trimming curve and the intersection curve is relatively simple—use the bivariate Sturm sequence again. It is also relatively simple to find the corresponding points on the other patch—the trimming curve has a surface associated with it, and this surface, when intersected with the second patch, gives another curve in the domain of that second patch. From this we obtain the intersection points on the intersection curve in the second patch, and figure out which points correspond by matching the intervals in 3-space. Notice that if we consider the patch boundaries as being trimming curves themselves, then this method can also be used for clipping to the patch boundary (section 4.2). Here, again, matching intervals in 3-space will be far more efficient than using multivariate Sturm sequences. It is possible to determine bounds, based on patch curvature, for the 3-space region occupied by an interval. Given such bounds and using a complete list of all intersection points in real space (not just within the patch boundary), the 3-space matching technique will be exact.

The actual pruning step is carried out by determining the orientation (inside/outside) of one point (we choose the starting point on the intersection curve) using 2D ray-shooting (described in section 3.3.3). Propagating this information to adjacent sections of the curve clearly identifies which curve segments lie inside the trimmed region and which lie outside. Along with this intersection curve segment, we also maintain the patch number of the other solid that defined this curve. We use this information later when we update the topology information.

3.5 Merging intersection curves

In the previous step, we obtained the intersection curves on each patch for all patch-patch pairs. We now need to merge these curves (which will define the new trimming curves). This is performed by matching the endpoints of each of the intersection curves, thereby partitioning the patch domain into closed loops. Notice that the monotonic segments of each intersection curve are already merged. Basically, we must check both endpoints of each intersection curve against the endpoints of all of the other intersection curves. If we find that two curves share a common endpoint, then we store this information and consider the curves merged. Checking for point equality is implemented as a kernel routine. Checking for actual equality, however, is not necessary. If there is no degeneracy involved, then each endpoint will either lie on a trimming curve or will match up with exactly one other endpoint. To find this other endpoint, simply refine the rectangles bounding the endpoints until each

endpoint rectangle overlaps with at most one other endpoint rectangle. The overlapping rectangles then give the endpoints which match with each other.

Once the intersection curves have been merged, we need to once again check whether the bounding boxes of the monotonic segments are overlapping and, if so, subdivide the curve appropriately.

3.6 Partitioning trimming boundaries

Once all the intersection curves are merged within each patch, they will partition the trimmed domain. Otherwise, it is a degenerate intersection (we discuss such cases in section 6). Fig. 4(a) shows intersection curves inside a trimmed domain. $\mathbf{c_i}$'s (with endpoints $\mathbf{p_i}$ and $\mathbf{p_{i+1}}$) are monotonic curves (in both s and t) that form the trimmed boundary of the patch. $\mathbf{i_0}$, $\mathbf{i_1}$, and $\mathbf{i_2}$ are the intersection curves computed with various patches of the other solid. $\mathbf{t_0}$ is a turning point on the curve $\mathbf{i_2}$. Remember that all the turning points are identified when the topology of the intersection curve is resolved (see appendix, section 8.3). $\mathbf{q_i}$'s are points on the intersection curve where the curve intersects the trimmed boundary. Given this information, Fig. 4(b) shows the actual partitions ($\mathbf{R_i}$'s). To compute the explicit B-rep of the resulting solid, each of these partitions is generated. We now present an algorithm that computes these partitions provided the intersection curves have no singularity (self intersection or vanishing tangent) in the trimmed domain.

The main idea in this algorithm is that since the intersection curve segments ($\mathbf{i_0}$ and $\mathbf{i_1}$ in Fig. 4(c)) do not cross each other, each resulting partition starts at one endpoint of a curve segment and ends at the other endpoint of the same curve segment. We number the endpoints of the intersection curve segments such that $\mathbf{q_{2j}}$ and $\mathbf{q_{2j+1}}$ belong to $\mathbf{i_j}$. The algorithm works in three steps.

- Each endpoint of a curve segment (for example, $\mathbf{q_0}$ of $\mathbf{i_0}$) lies on a unique curve (except when it coincides with one of the curve endpoints of the boundary) of the trimming boundary. In fact, points like $\mathbf{q_0}$ are determined as the intersection of $\mathbf{i_0}$ with $\mathbf{c_0}$. Note that even though $\mathbf{c_0}$ and $\mathbf{c_1}$ could be part of the same algebraic curve, the association of $\mathbf{q_0}$ with $\mathbf{c_0}$ is determined based on the monotonicity of the $\mathbf{c_i}$'s. Each boundary curve $\mathbf{c_i}$ is then partitioned into multiple segments depending on the number of $\mathbf{q_j}$'s lying on it.
- This is followed by a traversal of the trimming boundary in a consistent order by maintaining a stack. Two types of elements are pushed in the stack—curve segments, and curve endpoints. Initially, we keep pushing in the boundary curve segments until we reach a vertex like $\mathbf{q_0}$. Let the vertex number be k. If the topmost curve endpoint type of the stack (say, $\mathbf{l}$) has a

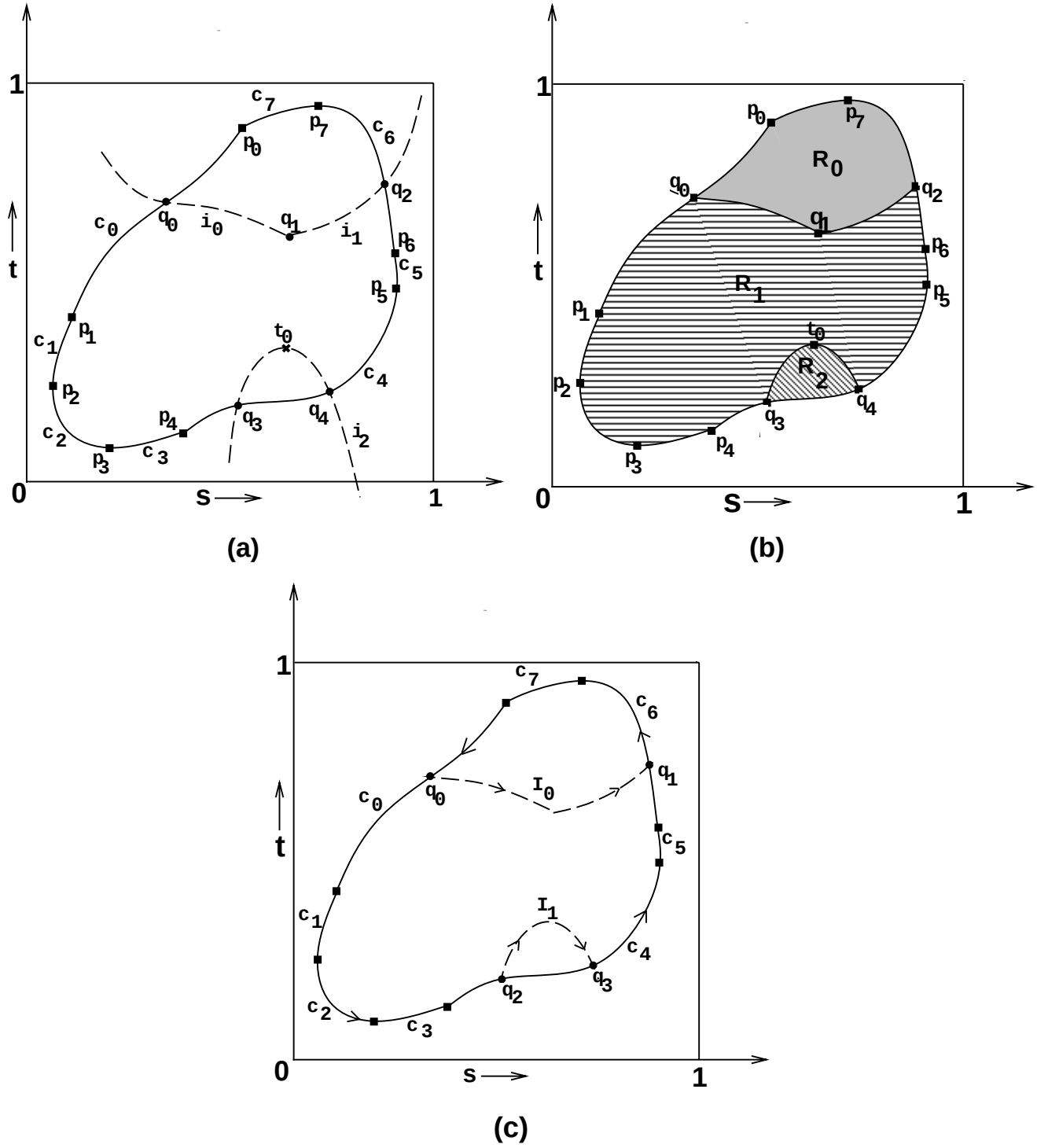


Fig. 4. (a) Intersection curves inside trimmed domain (b) Partitions introduced by intersection curves (c) Partitioning a trimmed patch with chains of algebraic curves

number $(k+1)$ or $(k-1)$, then a partition has to be read out. Otherwise, vertex k is pushed into the stack followed by all the curves that comprise $i_{\lfloor k/2 \rfloor}$. If a decision to read out a region has been reached, all the curve segments until vertex $\mathbf{l}$ are popped. Curves comprising $i_{\lfloor k/2 \rfloor}$ are pushed again because they are required by the next region too. The curve segments must be pushed into the stack in an order such that when they are read out again, the curve will be oriented consistently.

- Till now, we have considered only intersection curve segments whose endpoints lie on the trimming boundary. However, there may be loops that lie completely inside the boundary. Any loop is present (if at all) inside one of the obtained partitions. Each of the loops (starting from the innermost if the loops are nested) themselves form a partition. The remaining part of the region (it has boundaries with multiple components) is broken into simple regions by introducing a simple cut from the loop to the boundary of the partition or the next loop.

This completes the algorithm to compute the partitions introduced by intersection curves. A feature of this algorithm is that the adjacency information between the various partitions (which is necessary to avoid redundant, expensive ray-shooting queries during component classification) is obtained by the order in which they are read out.

3.7 Updating topological information

It is clear from the previous section that intersection computation introduces new vertices, edges, and faces in the solid. This change needs to be incorporated in our topological structure. This information about the adjacency between the various faces will significantly reduce the component classification time (see 4.8). At this time, we just concentrate on the face adjacency. Vertex and edge information is updated during final solid generation.

The new graph is a refinement of the original adjacency graph. Remember that a graph vertex corresponds to a face of the solid. Each graph vertex in the original graph is split into a few graph vertices depending on the partitions obtained due to the intersection curves. For example, if a patch was partitioned into three patch components, the graph vertex corresponding to the original patch would now be split into three graph vertices. We need to figure out the adjacency relationship between the newly created graph vertices. Consider, for example, that graph vertices $\mathbf{u}$ and $\mathbf{v}$ were adjacent in the original graph. Due to the intersection curves, let the graph vertex $\mathbf{u}$ be split into $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_m}$, and let the graph vertex $\mathbf{v}$ be split into $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_n}$. The adjacency between the various $\mathbf{u_i}$'s (similarly $\mathbf{v_i}$'s) has already been determined (during partitioning). These adjacencies (let us call it set S) are not considered and

thus the graph edges representing those adjacencies are purposefully left out in the new graph. Let e be the edge along which $\mathbf{u}$ and $\mathbf{v}$ were adjacent in the original graph, and let it be divided into k portions during partitioning. Then all the adjacencies between $\mathbf{u_i}$'s and $\mathbf{v_j}$'s can be obtained in $O(k)$ time. The number of connected components in this graph gives the number of solid components introduced by the intersection curves. Let the solid components be named $CC_0, CC_1, \ldots$. Note that each CC_i has a collection of faces.

We introduce some notation here in order to discuss how we obtain the connectivity between the various CC_i's. Let R be a mapping which takes a graph vertex in the new graph to the corresponding graph vertex in the original graph. For example, if $\mathbf{u}$ was split into $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_m}$, then $R(\mathbf{u_i}) = \mathbf{u}$. Two solid components CC_i and CC_j are connected if

$$\{\exists \mathbf{u_1} \in CC_i, \exists \mathbf{u_2} \in CC_j | R(\mathbf{u_1}) = R(\mathbf{u_2}) \; and \; (\mathbf{u_1}, \mathbf{u_2}) \in S\}$$

Using this, we have obtained the various surface components and their connectivity. Next we will resolve each of these surface components (inside/outside) with respect to the other solid.

3.8 Component classification

Component classification involves determining whether a given surface component of one solid is inside or outside the other solid. It is obvious that the entire surface component (as obtained in the previous section) lies completely inside or outside the other solid. In most polyhedral modelers, surface component classification is carried out locally [6]. When dealing with curved surfaces, though, the same technique cannot be used. The most general method used instead is based on ray-shooting. Ray-shooting is done by firing a semi-infinite ray in an arbitrary direction and checking for intersections with the other solid. If the number of intersections is even, the point (and hence, the entire surface component) lies outside the solid; if it is odd, it lies inside.

There are three steps involved in our algorithm to perform component classification. The first step involves getting a point that is part of the surface component. This reduces to finding a point inside the trimming boundary of one patch. This is accomplished by a simple 2D line intersection, as follows. We initially choose some rational point $p = (s, t)$ in the domain such that t lies between the lower and upper extents of the trimming boundary. A horizontal line passing through p (in both directions) is intersected with the boundary, and all the intersections are determined using univariate root isolation. The intersections must be even in number and are of the form $(s_1, t), (s_2, t), \ldots, (s_{2n}, t)$. The s_i's are algebraic numbers and are represented as small rational inter-

vals. Choosing the midpoint of the bounding intervals of s_{2i-1} and s_{2i} for $i = 1, 2, \ldots, n$ gives one rational point inside the trimming boundary. Let this point be called **q**.

The second step involves actual ray-shooting in 3-space. To perform 3D ray-shooting, we compute **q**'s mapping in 3-space using the patch parametrization. Let this point be (x_q, y_q, z_q). This is a rational point because of the formulation of the patch parametrization. We pick a random direction and fire a semi-infinite ray in that direction (i.e. create any line through q and solve for intersections on one side of the point). We compute all the intersections of this ray with each patch of the other solid. This is done as in the curve-surface intersection computation described earlier in this section (4.2). However, not all the intersection points computed this way lie inside the trimmed boundary of the patch. Checking whether the intersection point lies inside the trimmed boundary, as in section 3.3.3, is the third step of component classification.

3.9 Final b-rep generation

The trimmed patches that make up the final solid are determined by the Boolean operation performed. Given two solids $solid_1$ and $solid_2$, we decide on the final B-rep depending on the Boolean operation as follows. Recall that the word "component" refers to surface patches, not to solid elements.

- *Union:* All components of $solid_1$ that lie **outside** $solid_2$, and vice-versa are retained.
- *Intersection:* All components of $solid_1$ that lie **inside** $solid_2$, and vice-versa are retained.
- *Difference:* All components of $solid_1$ that lie **outside** $solid_2$, and all components of $solid_2$ that lie **inside** $solid_1$ are retained.

We also update the topology information. Each connected component that is retained in the final solid has some graph vertices (faces of the solid) whose complete adjacency is not determined. These correspond to edges which are formed by intersection curves. The graph vertex that should be adjacent to this graph vertex along this edge in the final graph is part of the other solid, and is the surface that formed the intersection curve. We have maintained the patch number of the other solid that resulted in an intersection curve, and use it to complete the adjacency. From this graph, all topological information is computed.

4 Analysis and performance

In this section, we shall discuss the theoretical and empirical complexity for some important steps of our algorithm. The most dominating steps in terms of time are the root isolation of bivariate polynomials, and the topological resolution of intersection curves. We optimize these algorithms, and implement them as part of the kernel routines. The complexity of steps involving partitioning based on the intersection curve, and the face connectivity generation are very small compared to the total cost, and their analysis is omitted here.

4.1 Worst case analysis

Root isolation: Most of the results involving real root isolation are based on Sturm sequences, and we quote a result from Davenport [4] for the worst-case time complexity of root isolation algorithm for univariate polynomials.

Theorem 1 *[4] The running time of the root isolation algorithm based on Sturm sequences of univariate polynomials is bounded by* $O(n^6(\log n + \log \Sigma a_i^2)^3)$, *where* n *is the degree of the polynomial and* a_i *are its coefficients.*

But this bound is too pessimistic, and a result based on [5] predicts the average case to be more like $O(n^4)$.

Sylvester resultant: We shall now look at the growth of the coefficient size while computing the Sylvester resultant of two polynomials. Let

$$f^n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \tag{2}$$

and

$$g^m(x) = b_m x^m + b_{m-1} x^{m-1} + \ldots + b_1 x + b_0 \tag{3}$$

be the two polynomials whose resultant we are computing. In our case, this resultant is used to compute the volume function, and hence the a_i's and b_j's are symbolic coefficients (polynomials in u, s, and t as described in section 2.1). The resultant of these two polynomials is an $(m + n) \times (m + n)$ matrix. Let

$$c = \max\{\max_{i=0}^{n}(wt(a_i)), \max_{j=0}^{m}(wt(b_j))\}$$

be the maximum coefficient size of the polynomials, where $wt(a_i)$ is the sum of the absolute value of the coefficients of a_i.

The resultant (which is the determinant of the Sylvester matrix) has maximum weight bounded by W_{m+n}, where W_{m+n} is given by the recurrence

$$\begin{aligned} W_{m+n} &= (m \ + \ n) \ c \ W_{m+n-1} \\ &\leq [(m \ + \ n) \ c]^{m+n} \end{aligned}$$

For the case of quadrics, tori, and the other common low-degree solids, m and n are each at most four. Thus the bit complexity of the coefficients of the univariate polynomial is roughly 8 times the original bit complexity in the worst case.

Topological resolution of intersection curves: The algorithm described [1] computes the sign invariant decomposition of an algebraic curve in a finite domain. The time complexity is given by the following result:

Theorem 2 *[1] Given a bivariate polynomial of total degree n and coefficient size d in E^2, it is possible to obtain a sign invariant decomposition of the curve in time $O(n^{12} \ (d \ + \ \log n)^2 \ \log n)$.*

We emphasize that these are worst-case bounds. In most practical cases, the complexity is not nearly as bad.

4.2 Improving performance of exact arithmetic

As exhibited by the worst case bounds, the arithmetic of these computations can be expensive. Even though, in reality, we do not experience such a drastic increase in bit complexity for intermediate computation, it nevertheless grows. The use of symbolic arithmetic also causes tremendous efficiency problems. For this reason we have looked into several areas for improving the overall efficiency of our exact arithmetic-based approaches.

Sturm sequence calculations: The multivariate Sturm sequence computation is the key underlying computation in our overall approach. The overall running time of the approach is governed primarily by the efficiency of the Sturm sequence computation. As mentioned previously, multivariate Sturm sequences are used to deal with vertices of the solids. These vertices are represented in the patch domain as the intersection of two algebraic plane curves within some interval. Multivariate Sturm sequences allow us to count the number of roots within a two-dimensional rectangle. Since the plane curves may intersect at several points within the patch domain, it is necessary to isolate

the various intersection points from each other. This can be done by counting the number of intersections within the $[0,1]$ x $[0,1]$ patch domain, and repeatedly subdividing the rectangle into smaller and smaller rectangles until any rectangle has at most one intersection point inside of it. In the same way, a rectangle may be reduced in size by repeatedly subdividing and choosing the subrectangle which still contains the intersection point.

Since the Sturm sequence calculations comprise most of the time spent in the overall Boolean operation, it is important that they be as efficient as possible. Several approaches are used to make our implementation of the multivariate Sturm sequence computation more efficient. [Please refer to section 2.1, for notation used here.] First, it should be noticed that f_3 is easily solved for a_1 (or a_2). This value can then be substituted into f_1 and f_2, eliminating two of the resultant computations in the formula for the volume function. Secondly, the numeric values for s and t can be plugged in ahead of time. This greatly simplifies the computation of the volume function and Sturm sequence by reducing the amount of symbolic arithmetic which must be performed. Even though the volume function must be computed four times for a single rectangle, the savings due to not performing symbolic arithmetic far outweigh the additional cost. At this point, we see that the volume function becomes:

$$V(u) = \frac{Res_{a_2}(\overline{f_1}(u,a_2), \overline{f_2}(u,a_2))}{u^{deg(f_1(s,0))deg(f_2(s,0))}},$$

where $\overline{f_1}(u,a_2)$ is $f_3(u,a_1,a_2)$ solved for a_1 and substituted in to $f_1(a_1,a_2)$.

In order to eliminate a_2 from $\overline{f_1}$ and $\overline{f_2}$, we use a multivariate interpolation approach similar to one in [12]. We know that the final answer will be a polynomial in u, and set up a Vandermonde system to solve for the final polynomial. In this way, all symbolic arithmetic is eliminated from the resultant computation. Because of their regular structure, Vandermonde systems can be solved in $O(n^2)$ time. In practice, however, we have found that Gaussian elimination is faster than the asymptotically superior methods for solving the Vandermonde system, for the matrices that we deal with (maximum size about 64).

Further optimizations which can be made to Sturm sequence evaluation include the use of polynomial remainder sequences and incorporation of hybrid floating-point/exact methods to increase efficiency while maintaining exactness. The incorporation of floating-point numeric methods appears to be the most promising avenue for increasing efficiency further. Some of our results in doing this are presented in [10].

Another approach to speed up the computations is the use of finite field computation. In this approach, all computation can be performed over finite fields, which is fast since computer hardware is used directly. Then a probabilistic algorithm, based on the Chinese Remainder Theorem, will be used to recover

the actual coefficients. This approach is like that used in [12]. Finite fields are especially useful when the final number to be computed is small relative to the size of the numbers in intermediate computation, such as in resultant computations. The time complexity of the resultant computation is directly proportional to the number of primes used in the finite field computation. To reduce this number, we use primes of maximum possible magnitude. Most current implementations of *bignum* libraries use finite fields of order 2^{16} to prevent overflow when taking products. Most of the current machines provide multiplication instructions that give the result out in two registers. Taking advantage of this fact, we use an assembly level subroutine that performs multiplication. This allows us to use finite fields of order as high as 2^{31}. Compared to finite fields of order 2^{16}, we get almost two-fold improvement in speed.

Lazy evaluation during root isolation: Theorems exist which predict the number of bits of precision necessary in the worst case to isolate the roots of polynomials (e.g. [3]). In general, however, these bounds are more pessimistic than is the case in practice. Another optimization that we perform to improve our speedup is to evaluate rational intervals (in root isolation) as lazily as possible. This is based on the assumption that the worst case bounds that govern the closeness of roots actually occur very rarely in practice. Thus, most of the time, we are able to isolate the roots of polynomials inside the domain of surfaces quickly. However, during later computation of other roots, it might be necessary to make sure that two roots are not the same (because of large overlapping intervals). At this time, we refine the computed intervals further to isolate the intervals for the two different roots. This lazy approach to root isolation behaves like an output-sensitive algorithm.

4.3 Empirical performance

A number of the basic routines necessary for this approach have been implemented. The routines were implemented on top of the LiDIA library [2], which supports multiprecision integer and rational number arithmetic. The performance of our code is, to a large degree, governed by the performance of the underlying rational arithmetic library. We should emphasize that although we have implemented various steps of our approach, they currently have not been integrated into a complete system.

Among the implemented portions of the code are the routines for polynomial manipulation, resultant computation, Sturm sequence generation, and 2-D root finding. We have worked at optimizing these routines by incorporating floating-point and other speedups. Some of these speedups are described in [10], and we continue to work on others. We should emphasize that the timings presented here are not optimized and are presented to highlight some

Case	Precision	# Sturm	Vander. Build (sec)	Vander. Solve I (sec)	Vander. Solve II (sec)	Polynomial Division (sec)	Total Time (sec)	Avg. Time (sec)
1	0.01	117	11.76	12.88	5.88	0.62	19.44	0.166
	0.001	187	20.83	18.98	8.86	1.36	32.71	0.175
	0.0001	262	38.04	60.97	16.37	7.97	67.53	0.258
	0.00001	336	45.13	41.10	19.37	3.55	71.48	0.212
	0.000001	395	61.58	49.67	22.52	4.92	93.00	0.235
2	0.01	35	58.32	27.11	18.46	12.92	90.73	2.59
	0.001	54	114.54	53.27	34.08	30.91	181.36	3.35
	0.0001	76	207.03	79.63	48.55	60.61	319.10	4.19
	0.00001	92	290.40	108.16	66.51	91.05	451.84	4.91
	0.000001	110	436.10	146.03	81.51	120.51	643.07	5.84
3	0.1	19	250.16	305.34	257.47	3733.10		
	0.01	35	576.42	650.14	519.58	7979.37		

Fig. 5. Timing results for a basic (non-optimized) curve-curve intersection implementation.

of the computational problems faced. These timings do *not* contain optimizations such as those described in [10]. Such optimizations have improved the speeds for all computations shown here by *more* than an order of magnitude. Alternative approaches [7] have yielded more than two orders of magnitude speedup.

We give here some timing results for a basic implementation of a curve-curve intersection routine. The results are presented in Fig. 5, and the program was run on an an SGI R10000 200 MHz processor. The machine was loaded at the times these tests were made, and the variable load accounts for some of the differences and abnormalities in the timing (e.g. in the third run listed). These timings are *not* intended to be exact specifications of the time that would be required in an actual implementation, but rather to give a feel for the bottlenecks and relative speeds of various parts.

The various columns stand for the following (all times are in seconds):

- **Case:** Which two curves are being intersected (more details later)
- **Precision:** The precision to which each intersection point was determined. The number refers to the maximum width in s and t of the interval bounding

the intersection point(s).

- **# Sturm:** The number of multivariate Sturm sequence calculations involved. Each Sturm sequence computation involved building a Vandermonde matrix, solving the matrix to get a single polynomial, and then determining the univariate Sturm sequence for that polynomial. Recall that to count the number of intersections of two curves within one box requires four multivariate Sturm sequence computations.
- **Vander. Build:** The time taken for all of the Vandermonde systems to be built. One is built for each multivariate Sturm sequence computation. Building the Vandermonde system involves finding the determinant of several Sylvester matrices with rational matrix entries.
- **Vander. Solve I:** The time taken to solve the Vandermonde systems using the $O(n^2)$ Vandermonde-specific algorithm.
- **Vander. Solve II:** The time taken to solve the Vandermonde systems using Gaussian elimination.
- **Polynomial Division:** The time taken to generate the univariate Sturm sequence for the polynomial determined by the Vandermonde system solution. Recall that this follows the procedure of Euclid's algorithm for finding the GCD.
- **Total Time:** The total amount of time taken for all computation, assuming Gaussian elimination was used to solve the Vandermonde system.
- **Avg. Time:** The average time per multivariate Sturm sequence computation. [Total Time/# Sturm].

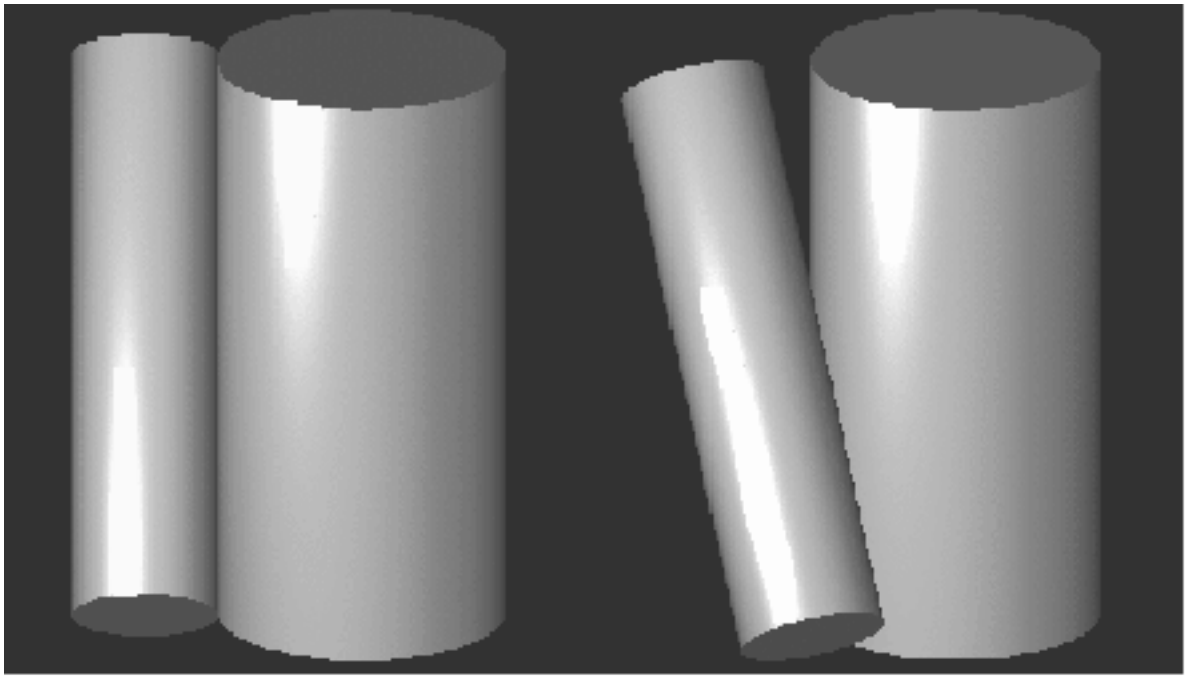
The three cases timed are as follows:

- **Case 1:**
  $4s^2 - 4s + t = 0$
  $s + 5t^2 - 5t + \frac{1}{8} = 0$
  These curves intersect at four points inside of an interval of size 1 in s and t. The test isolates all four points and then determines each intersection point to the indicated precision. The computation uses a Vandermonde matrix of size 10×10, to compute a polynomial of degree four. [This implies that the Vandermonde system could have been much smaller, however as mentioned, these implementations are not optimized. This is true for all three cases.]
- **Case 2:**
  $$-14400x^4t^2 - 28800s^2t^2 - 14400t^2 - 38400s^3t + 14400s^2t - 38400st + 14400t - 59501s^4 - 45002s^2 + 19200s - 3501 = 0$$
  $$-57600s^3t^2 - 57600st^2 - 115200s^2t + 28800st - 38400t - 238004s^3 - 90004s + 19200 = 0$$
  These curves (from the example given earlier) intersect at one point inside the region of size 1 in s and t. The test determines this root to the indicated precision. The computation uses a Vandermonde matrix of size 23×23 to determine a polynomial of size 8.
- **Case 3:**
  $t^4 - \frac{4}{7}t^3 + \frac{6}{49}t^2 - \frac{4}{343}t - 100s^4 - 500s^3 - 17s^2 - 53s - \frac{2320}{194481} = 0$

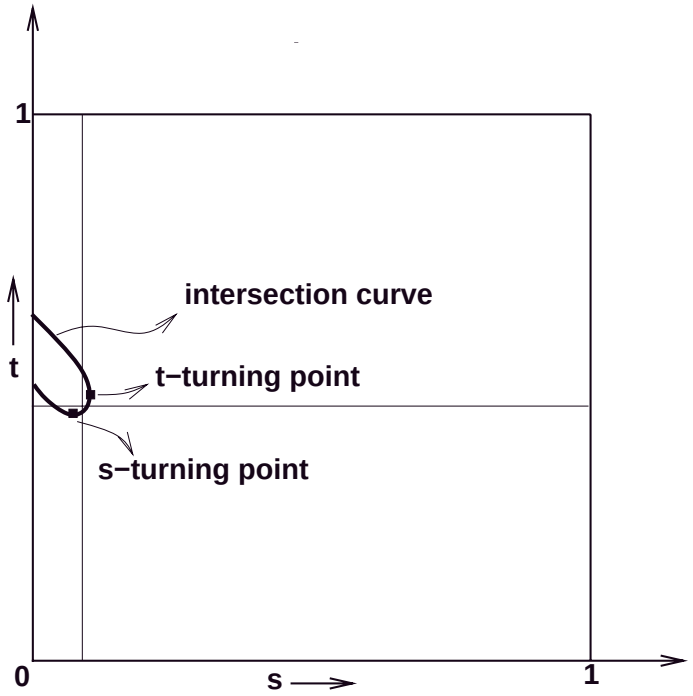
$t^4 - \frac{4}{5}t^3 + \frac{6}{25}t^2 - \frac{4}{125}t - \frac{1}{8}s^4 + \frac{3}{4}s^3 + \frac{1}{8}s^2 - \frac{1}{4}s - \frac{623}{1250} = 0$
These curves intersect multiple times, but the test was only to determine one root to the indicated precision starting from an interval of size 1 in s and t. The Vandermonde matrix here was size 65×65 for a polynomial of degree 16.

From these timing results, we can draw several conclusions:

- For the size of Vandermonde systems we are dealing with, Gaussian elimination appears to work better than the $O(n^2)$, Vandermonde-specific, algorithm does. For the larger Vandermonde systems, the difference becomes less, but Gaussian elimination is still faster.
- The average time per multivariate Sturm sequence computation increases for more precise computations (with the exception of one case, probably due to varying processor load). This is to be expected, since higher precision requires more bits to be used in each computation. For exact arithmetic, basic arithmetic operations do not take constant time, but rather take longer when more bits of precision are required. The results seem to indicate that the extra time due to increased precision, while a significant factor in the overall time taken, is not excessive.
- Time for polynomial division to generate the Sturm sequence is not very significant for low-degree polynomials, but becomes the dominating factor in Sturm computations involving higher degree polynomials. The reason for this increase is threefold:
  - The starting polynomial is larger, so each individual polynomial division takes longer on average (i.e. there are more terms to deal with).
  - Since the starting polynomial is of higher degree, the Sturm sequence itself is longer. Generally, the Sturm sequence will be a sequence of n polynomials, where n is the degree of the original polynomial.
  - The size of the coefficients grows larger very rapidly as the number of terms in the Sturm sequence grows. When we looked more closely at the time taken for polynomial division, it appeared that this was, by far, the most significant cause of slowdown. This coefficient growth has been well studied in the context of polynomial GCDs, and the techniques applied there (such as polynomial remainder sequences) can also be applied here. Our implementations of this optimization do indeed provide a significant speedup [10].



(a)



(b)

Fig. 6. (a) Two views of just interpenetrating cylinders (b) Complete intersection curve in the domain of one patch

5 Example

5.1 Example

In this section, we illustrate some key parts of our approach using an example of two cylinders which are just interpenetrating (see Fig. 6(a)). The cylinders are of radius 1 and 0.5, and their centers are spaced 1.49 units apart. The cylinders are rotated with respect to each other. We divide the surface of each cylinder into four equal parts and represent each of them as a rational parametric surface with rational coefficients. The parametric form of a sample patch from each cylinder are given below.

$$
\begin{aligned}
X(s,t) &= 1 - s^2 \\
Y(s,t) &= 2\ s \\
Z(s,t) &= 2\ t + 2\ s^2\ t - 1 \\
W(s,t) &= 1 + s^2
\end{aligned}
$$

$$
\begin{aligned}
\overline{X}(u,v) &= \tfrac{199}{100}\ u^2 + \tfrac{99}{100} \\
\overline{Y}(u,v) &= \tfrac{4}{5}\ u - \tfrac{6}{5}\ v - \tfrac{6}{5}\ u^2\ v + \tfrac{3}{5} \\
\overline{Z}(u,v) &= \tfrac{3}{5}\ u + \tfrac{8}{5}\ v + \tfrac{8}{5}\ u^2\ v - \tfrac{4}{5} \\
\overline{W}(u,v) &= 1 + u^2
\end{aligned}
$$

After implicitizing using Dixon's formulation, the implicit forms are

$$
\begin{aligned}
f(x,y,z,w) &= w^2 - x^2 - y^2 \\
g(x,y,z,w) &= 19701\ w^2 - 29800\ x\ w + 10000\ x^2 + \\
&\quad 6400\ y^2 + 9600\ y\ z + 3600\ z^2
\end{aligned}
$$

To obtain the intersection curve of the two patches in the domain of the first patch, we substitute its parameterization into the implicit form of the second patch ($g(x,y,z,w) = 0$).

$$
\begin{aligned}
h(s,t) = {} & -3501 + 19200\ s - 45002\ s^2 - 59501\ s^4 + 14400\ t - 38400\ st + 14400\ s^2t - \\
& 38400\ s^3t - 14400\ t^2 - 28800\ s^2t^2 - 14400\ s^4t^2 = 0
\end{aligned}
$$

Since the patches are untrimmed, we have to compute the starting points of the curve on the boundary of the patch. Substituting $s = 0$ into h and computing the volume function for this univariate case, we get

$$
V(u,t) = -3501 + 14400\ t - 14400\ t^2 + 14400\ u - 28800\ t\ u - 14400\ u^2
$$

We computed the Sturm sequence of this volume function, and isolated the roots of the original equation between $[0,1]$ to within a precision of $\frac{1}{100}$. The

two roots were

$$\left[\frac{226834}{390625}, \frac{1838}{3125}\right], \quad \left[\frac{1298}{3125}, \frac{6634}{15625}\right]$$

These numbers give the ranges in t for which there is an intersection of the intersection curve with the s=0 edge of the patch domain. Of these, only the first one corresponds to a point inside the domain of the second patch. This was obtained by inversion and a bivariate Sturm sequence generation. The point corresponding to $\left(0, \left[\frac{226834}{390625}, \frac{1838}{3125}\right]\right)$ inside the domain of the second patch is $\left(\left[\frac{7352}{78125}, \frac{64}{625}\right], \left[\frac{43682}{78125}, \frac{8866}{15625}\right]\right)$. The s (or t) turning points on the intersection curve were obtained by performing bivariate Sturm sequence root isolation on the pairs of polynomials $h(s,t)$ and $h_s(s,t)$ $(h_t(s,t))$. The s and t turning points were found to be $\left(\left[\frac{20464}{390625}, \frac{4352}{78125}\right], \left[\frac{5764}{15625}, \frac{146044}{390625}\right]\right)$ and $\left(\left[\frac{1096}{15625}, \frac{248}{3125}\right], \left[\frac{2}{5}, \frac{6346}{15625}\right]\right)$ respectively.

Now that we have the turning points of the intersection curve, we compute the topological resolution. After separating the turning points and evaluating the signs of slope at various grid lines, the connectivity of the curve is obtained unambiguously. The resulting curve in the domain of the first patch is shown in Fig. 6(b). The intersections of the curve with the s=0 axis, the turning points in s and t, and the grid lines for curve connectivity computation are all shown in the picture. Note that the curve shown was obtained by intersecting this patch with *all* the patches of the second cylinder.

6 Conclusion

In this paper, we have presented an approach for computing B-reps for Boolean combinations of low-degree solids specified with rational parametric surfaces, using exact arithmetic. We use exact arithmetic to perform reliable computations on a number of kernel routines, upon which the rest of the modeler is built. The efficient and accurate implementation of the kernel routines allows us to have an efficient and reliable method for the overall B-rep computation, maintaining an exact representation throughout. By eliminating some robustness problems (those due to numerical errors), our method is a step toward a robust and accurate system.

There are a number of avenues for future work in this area, some of which have already been discussed in our previous paper [8].

6.1 Acknwoledgements

We would like to thank the reviewers for their many helpful comments and suggestions on the content and style of this paper.

References

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

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

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

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

[5] L. E. Heindel. Integer arithmetic algorithm for polynomial real zero determination. *Journal of ACM*, 18(4):535–548, 1971.

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

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

[8] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate b-rep generation of low degree sculptured solids using exact arithmetic: I - representations. *Computer Aided Geometric Design.*

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

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

[11] S. Krishnan and D. Manocha. An efficient surface intersection algorithm based on the lower dimensional formulation. *ACM Transactions on Graphics*, 16(1):74–106, 1997.

[12] D. Manocha and J.F. Canny. Multipolynomial resultant algorithms. *JSC*, 15(2):99–122, 1993.

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