Robustness in Geometric Computations*

Christoph M. Hoffmann†
Computer Science
Purdue University

April 1, 2001

Abstract

Geometric computation software tends to be fragile and fails occasionally. This robustness problem is rooted in the difficulty of making unambiguous decisions about incidence and nonincidence, fundamentally impairing layering the geometry software reliably. Additionally, geometric operations tend to have a large number of special and singular cases, further adding to the difficulty of creating dependable geometric software. We review the problem origins and ways to address it.

1 Introduction

Good software development uses the fundamental strategies of layering and folding. The purpose of layering is to manage complexity and to achieve reuse of code components. The purpose of folding special cases is to achieve more compact code and to reduce thereby the opportunity for errors. These activities are so fundamental that most developers no longer think about them and consciously consider more the derived aspects such as complexities of code interdependence etc., as described for instance by Lakos [1]. In the development of geometric software, however, we are required to review the code development fundamentals as part of the algorithmic strategy, and must re-examine assumptions that have become automatic.

*Work supported in part by ARO Contract 39136-MA and by NSF Grant CCR 99-02025.
†Not a member of ASME

1.1 Ordinary Layering and Folding

In the case of polyhedral intersection, the code layers range from the very generic, usable by most geometric computations, to the specialized, tailored to particular ways of representing solids and carrying out Boolean operations on them. The layers might look as follows:

1. Memory management and basic system services, foundational data structures such as graphs and lists.

2. Simple operations such as vector addition and subtraction, inner product, ordering collinear points, and more.

3. Composite operations such as determining a plane from points, point on plane test, line/plane intersection, and so on.

4. Higher utilities such as face/face intersection, boundary traversal, topology management.

5. Component operations such as shell intersection, and, building on it, the intersection algorithm itself.

Proper structure lets us use much of these internals for other purposes as well, and related algorithms, such as polyhedral union and difference, are easily added at the higher levels. It goes without saying that the lower levels must be absolutely reliable. All contingencies that can arise during execution of those operations must be fully accounted for in the data structures and in the logic of the code.

For polyhedral intersection, a number of special cases should be folded into one. General strategies here might include using homogeneous coordinates that eliminate problems with infinite coordinates. Dimensional fusing can be used; for instance, an intersection method is laid out largely dimension-independent so that such cases as coplanar face intersections and containment tests can be treated recursively in the same way as shell intersection and point-in-solid tests. The strategies here are more specific to the data structures, coordinate conventions, and the nature of the geometric operations. When adding curved geometry, additional issues arise such as singularity treatment, single-edge face boundaries, and faces that topologically are not disks.

The bottom line is that the two strategic activities of layering and folding, so useful in code development in general, are not as easy to implement in the case of geometric computations. The problem arises from the close interaction of

numerical computation, on the one hand, with the logic of geometric reasoning and with the data structures that reflect the conclusions of that reasoning, on the other hand. This interplay is the source of difficulty for geometric computation. The more intricate the interaction between numerics and reasoning, and the more sophisticated and informative the data structures, the more arduous is the software development and the less reliable the resulting codes tend to be. Basically, the lower layers of the code remain fragile and not fully dependable.

CAD systems are among geometry software with the highest complexity. They are used widely by designers and manufacturing engineers in discrete product design. By now, those systems have become extremely large measured by the functionality offered to the user, and the CAD houses have consolidated to a handful because of the complexity and cost of evolving and maintaining those systems. The robustness of CAD systems varies, and is achieved by the software houses, to the extent observed, by an elaborate and proprietary set of heuristics, that have been developed mostly by trial and error and may no longer be clearly understood.

The academic community began to consider the robustness problems of geometric computations in the 1980s. One of the earliest papers was Hopcroft and Hoffmann [2] in which the nature of the problem was clearly articulated. However, the problem was recognized well before then, although not necessarily formalized. Several general approaches can be distinguished, some addressing only the robustness problem, others trying to reduce, at the same time, the number of special cases and their complexities. The emerging consensus in academics is that exact computations offer the best hope of remedying fully the robustness problem. Unfortunately, this proposition is not practical in many situations, and the fact remains that the particular operation heavily influences the implementation details.

2 The Robustness Problem: Why and What

We illustrate a typical failure of a geometric computation, and then articulate the precise nature of the robustness problem. With this formulation we can measure what has been accomplished by a proposed approach. The example of failure is drawn from polyhedral intersection. Some concepts of solid representations are needed to understand the example and are explained first.

2.1 Representations of Solids

It is customary, though not necessary, to restrict to solids that are contained in a finite volume. When infinite objects are manipulated, they are usually limited to those infinite bodies that have a finite boundary.[1] Furthermore, the surface of a solid is required to be smooth in the topological sense, excluding, for example, fractal structures.

Modeling and manipulating solid bodies is one of the basic issues in geometry. There are implicit, unevaluated representations, such as CSG, which express a solid body symbolically using a few primitives and basic operations such as rigid body motion and the (regularized) Boolean operations of union, intersection and difference; e.g., Hoffmann [3], chapter 2. While such representations can be said to be fully correct in all cases, they simply defer the problem to the computations that interrogate this representation for rendering, location queries, mass properties, and the like. Still in use in some military systems, CSG has largely lost presence because of inherent inflexibilities when used for design of sculptured shapes.

An explicit boundary representation (Brep) represents the surface of the solid using vertices, edges and faces. Roughly speaking, a face might be a topological disk, an edge a topological segment, and a vertex a point.[2] Adjacency of the boundary elements is maintained by a graph structure. Variants of that structure are in use that embody different trade-offs. The conflicting requirements include on the one hand the desirability of minimal redundancy, and, on the other, the need to quickly access adjacent parts of the boundary. The variants also resolve differently the question whether an edge should be adjacent to exactly two faces, or whether there could be more adjacent faces. In the latter interpretation, separate volumetric components of the solid may share a common edge.

It is obvious that adjacency data is highly structured, and that algorithms should assume correct structure. After all, the structure is never manually defined, and always created by higher-level design operations carried out conscientiously. It is therefore a catastrophic event when adjacency data is inconsistent, and we now explain how such an inconsistency could arise. This would be a source of failure for a high-level algorithm manipulating a solid.

[1] An exception are algebraic half spaces defined as $f(x, y, z) \leq 0$. Their surface, $f(x, y, z) = 0$, may be infinite. A planar half space, $ax + by + cz + d \leq 0$, is a common object of this kind.

[2] But note that this convention necessitates more than one face to represent the curved surface of a cylinder and leads to artificial "seams."

2.2 How Intersection May Break

Suppose we want to intersect the two polyhedra shown in Figure 1. Conceptually, we intersect the two surfaces identifying edge loops that arise from face/face intersections. We partition the surface of the two polyhedra, and then sew together the relevant parts into a new surface. The sewing relies heavily on traversals of adjacency structures of the intersections. When we intersect the faces of the two solids, we will also locate the intersection points of edges of one solid with the faces of the other solid, because that is where a face/face intersection ends.

In our example, assume that the front edge of the tetrahedron cuts the top face of the cube at a steep angle and the front face of the cube at a shallow angle. When carried out numerically, it is therefore possible that we deem the edge intersection with the top face to be in the face interior, owing to the steep angle and the resulting better numerical conditioning of the linear system that determines the coordinates of the intersection. Nevertheless, the intersection with the front face may be deemed to lie on the edge of the face. After all, the shallow angle of the edge with the front face results in a linear system that has a poorer condition number and therefore yields coordinate values with a greater errors. The partition of the top and the front faces that would result from these incidence decisions is shown in Figure 2. The two partitions are logically inconsistent, because the edge of the tetrahedron now intersects the top face twice, at A and at B. This will break the sewing part of the algorithm in all likelihood, as the designer would not anticipate that a line intersects a plane in two separate points. The intersection algorithm would fail as a consequence.

Different strategies for determining the intersections lead to different types of failure. Also, there is the possibility that the sequence in which the intersections are made affects the outcome of the computation. Note that the face sequence in the data structure is arbitrary and not controlled.

2.3 Problem Formulation

The following captures the basic difficulty making geometric computations robust.

> *In geometric computations, logical facts such as incidence, separation, tangency, etc., are deduced based on numerical calculations. Further inferences are drawn from these deductions. The imprecision of floating-point arithmetic makes the conclusions drawn from the numerical calculations unreliable. In particular, the same logical question may arise repeatedly, in the form of different floating-point*

computations giving contradictory answers. Unless this problem is avoided, it is not possible to write fully correct/robust/reliable code for geometric operations using standard floating-point arithmetic.

Note that it is difficult recognizing that a particular geometric question has already been asked before in a different formulation. In our example, the question has been *"do two edges intersect?"* and that question has been answered inconsistently. While this particular question might seem to be easy to settle consistently in all cases, other questions might be hard to resolve consistently.

2.4 Tolerating Error

The severity of the robustness problem is a function of the intricacy of the data structures manipulated and the sophistication of the geometric reasoning implied. If we are to intersect simple structures such as line segments in the plane, we might very well tolerate incorrect answers some of the time if the objective is simply to find intersection points. When manipulating complex structures, such as the topology of solid boundaries, the tolerance for error is significantly diminished. In a sense, robustness problems grow with the evolution to more sophisticated geometric manipulations and reasoning. This is bad news because it impairs progress.

2.5 Solution Approaches

We can immediately identify the following strategies to address the robustness problem, partially or in whole:

1. *Use exact arithmetic*: This could be integer arithmetic, extended precision arithmetic, nonstandard, or symbolic arithmetic.

2. *Use symbolic reasoning*: Based on the nature of the geometric problem, we could symbolically reason, as part of the computation, which deductions have already been made, and which new ones need not be made anew, from floating-point computations, because they are a consequence of already known facts.

3. *Use reliable calculations*: Using interval arithmetic, we can provably enclose the result of an arithmetic calculation with a floating-point interval within which the result of the corresponding infinite-precision exact computation must lie.

Those approaches will be considered in turn.

3 Exact Arithmetic

If floating-point computations are the root problem, why not use exact arithmetic? More than 30 years ago, Macsyma implemented exact rational arithmetic, and we can surely too. There are two problems that complicate this reasoning:

1. *Proliferation*: If the input to a geometric operation has k-digit precision, the output may require higher precision.

2. *Irrationality*: Some operations result in coordinates that have no finite precision.

Both points can be demonstrated with simple examples.

For the proliferation problem, consider the intersection of two line segments. Assume a fixed precision with which the endpoint coordinates have been specified. Figure 3 shows two such segments. The grid line represent the resolution possible under the assumed precision. While some intersections are on grid points, thus have representable coordinates, others are not. That is, while the input points have a precision of k digits, the intersection points require a multiple of that precision to be representable. When we define other segments that connect such intersections, and we want to intersect them in turn, the precision needs to be increased anew to represent the second-generation intersection points. Iterating this process leads to an exponential growth in the required precision.

To see the irrationality problem, consider rotating the unit square, represented by vertex coordinates, by $45°$. The new vertex coordinates now involve radicals and are irrational.

Clearly, both problems can be solved in a general and expensive way, and such solutions are familiar from symbolic computation systems such as Mathematica, Maple, or Macsyma. However, how could we achieve the efficiency needed to be competitive with an unreliable floating-point implementation? Getting it right at the higher cost of exact arithmetic must ultimately find its justification in the requirements of the application. The straightforward way of implementing an infinite precision arithmetic package is rarely justified in practice.

Before dismissing exact computations from further consideration, let us consider studies that limit the needed precision in the context of specific applications.

3.1 Segment Intersection

We are given two segments by endpoints. Segment S_1 has the endpoints $A_1 = (x_1, y_1)$ and $B_1 = (x_2, y_2)$; segment S_2 has the endpoints $A_2 = (x_3, y_3)$ and $B_2 = (x_4, y_4)$.

3.1.1 Intersection Test

In the simplest case, we want to determine reliably whether S_1 and S_2 intersect, without computing the coordinates of the intersection. We describe an algorithm by Gavrilova and Rokne [4].

The points on the segment S_1 are given parametrically by $t_1 A_1 + (1 - t_1)B_1$ where $0 \leq t_1 \leq 1$. If the segments intersect, then the linear system[3]

$$t_1 A_1 + (1 - t_1)B_1 = t_2 A_2 + (1 - t_2)B_2$$

has a solution subject to $0 \leq t_1, t_2 \leq 1$. If there is a solution, it has the form

$$\begin{aligned} t_1 &= D_1/D_0 \\ t_2 &= D_2/D_0 \end{aligned}$$

where

$$D_1 = \begin{vmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} = 2\triangle_{(B_1,A_2,B_2)}$$

$$D_2 = \begin{vmatrix} 1 & x_4 & y_4 \\ 1 & x_2 & y_2 \\ 1 & x_1 & y_1 \end{vmatrix} = 2\triangle_{(B_2,B_1,A_1)}$$

$$D_0 = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} - \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_4 & y_4 \end{vmatrix} = 2(\triangle_{(A_1,B_1,A_2)} - \triangle_{(A_1,B_1,B_2)})$$

Note that the determinant values are proportional to the signed area of the triangle spanned by the points. To test that t_1 and t_2 have finite values and are in the required ranges we evaluate the sign of the determinants and suitable differences of them. That is:

1. If $D_0 = 0$, then the line segments are parallel or collinear. The subcases are

[3] A_k and B_k are vectors.

(a) If $D_1 \neq 0$, then the two segments are not collinear, hence do not intersect.

(b) If $D_1 = 0$ and $x_1 = x_2$; i.e., if the segments are vertical, then there is an intersection iff $y_1 \leq y_3 \leq y_2$ or $y_1 \leq y_4 \leq y_2$ or $y_3 \leq y_1 \leq y_4$ or $y_3 \leq y_2 \leq y_4$.

(c) If $D_1 = 0$ and the segments are not vertical, then compare the x-coordinates as in case (1b), or else the y-coordinates analogously, according to the slope of the line segments.

2. If $D_0 \neq 0$, then the segments are not parallel. Assuming that $D_0 > 0$, compute the value range of t_1 as follows:

```
if (D1 < 0) assert(t1 < 0);
else if (D1 == 0) assert(t1 == 0);
else if (D1 - D0 <= 0) assert(0 < t1 <= 1);
else assert(t1 > 1);
```

If $D_0 < 0$, then the code fragment must be suitably changed. The range of t_2 is determined analogously.

We observe that the decision process has been reduced to evaluating the sign of determinants and the sign of the difference of determinants. Because of the form of the determinants involved, all sign determinations are of the form

$$\sum x_k y_k - \sum x_j y_j$$

If we assume input coordinates that are single-precision floating-point numbers, the products $x_k y_k$ can be evaluated exactly with standard double-precision. The ESSA algorithm evaluates the sign of such a sum exactly.

3.1.2 ESSA Algorithm

We determine the sign of a sum of the form

$$s = \sum_{i=1}^{m} a_i - \sum_{i=1}^{n} b_i \qquad a_i, b_i > 0$$

where the individual terms are double-precision and positive. The underlying idea is to subtract the same quantity from a_1 and from b_1 in a manner that does

not change the sign of s. The terms in each sum are ordered from the largest to the smallest. The quantity u is chosen such that the differences $a_1 - u$ and $b_1 - u$ remain exact. Moreover, one of the terms is reduced to zero in one or more steps, so that eventually the sums are transformed such that one or both of them are empty. At that point the sign of s is known.

The algorithm is shown in Table 1; see [4, 5].

3.1.3 Computing Intersections

With input coordinates in single-precision, the extra effort expended by the ESSA algorithm achieves a reliable intersection test at modest cost. If we want to know the intersection coordinates, however, then we need to do more work. Now we require an exact summation of the products arising from the determinants, so the ESSA algorithm is not enough. Instead, we need to implement at the very least an exact summation of triple product terms.

Let $p = (x, y)$ be the coordinates of the point we seek, and assume that the segments intersect transversally. We will deal with the collinear case later. The the point p must satisfy both implicit line equations which, in terms of the segment end points are

$$\begin{aligned}(y_2 - y_1)x + (x_1 - x_2)y &= x_1y_2 - x_2y_1 \\ (y_4 - y_3)x + (x_3 - x_4)y &= x_3y_4 - x_4y_3\end{aligned}$$

The solution of the system is

$$\begin{aligned}x &= D_1/D_0 \\ y &= D_2/D_0\end{aligned}$$

where

$$D_1 = \begin{vmatrix} x_1y_2 - x_2y_1 & x_1 - x_2 \\ x_3y_4 - x_4y_3 & x_3 - x_4 \end{vmatrix}$$

$$D_2 = \begin{vmatrix} y_2 - y_1 & x_1y_2 - x_2y_1 \\ y_4 - y_3 & x_3y_4 - x_4y_3 \end{vmatrix}$$

$$D_0 = \begin{vmatrix} y_2 - y_1 & x_1 - x_2 \\ y_4 - y_3 & x_3 - x_4 \end{vmatrix}$$

We see that the quotients are formed from sums of triple products $x_ix_jy_k$ and $x_iy_jy_k$. Each sum has up to 8 terms. Therefore, if the input precision is k bits, the

sums require $3k+3$ bits to be correctly evaluated when given in integer arithmetic. Note that the integer grid of representable points is uniform, as shown in Figure 3, but the grid of representable floating point numbers is distinctly different, as seen in Figure 4.

Extended precision is needed to evaluate triple products exactly. If the exact inner product for double-precision is implemented, then the summation of triple products of single-precision floating point numbers can also be evaluated exactly.

Assume that we want to test whether two computed segment intersections are identical. If we work with rational coordinates, i.e., with integers as line coefficients, then the comparison can be made by comparing numerator and denominators. If the coordinates are equal, then the reduced quotients must have equal denominators and numerators. If we work with floating-point coordinates, then the accumulator structure explained next can be used.

We excluded collinear intersections thus far. Aside from the trivial cases of horizontal and vertical segments, we can first test collinearity as explained before, and then locate the contained endpoint(s) by comparison of one of the coordinates.

3.1.4 Exact Inner Product

An IEEE single-precision floating point number has an 8-bit exponent and a 24-bit mantissa. In the first implementation, we convert all legal single-precision values into a fixed-precision mantissa for which roughly 280 bit integers suffice. Allowing some additional bits to account for multiplications and additions within reasonable length, an accumulator of some 40 bytes can compute an inner product of moderate length accurately in almost all practical cases. The value held can be extracted again into a single-precision floating-point value with rounding. So, the intersection coordinates can be obtained as the quotient of two floats that are accurate to 23 bits, yielding a result that is accurate to at least 22 bits mantissa length.

There is no trick to implementing the exact inner product in this way. The most expensive operation is the initialization; adding a product and extracting a value is straightforward. Additional variables can be used to identify the nonzero portion of the accumulator.

There is another implementation that uses more space for the accumulation that is due to Kobbelt. It is appropriate when there is a lot of summation with only occasional value extractions. The idea is to defer the addition of the summation terms until a value needs to be extracted. The accumulator is an array A with size equal to twice the exponent range. Initially, each entry of A is zero.

When adding a number to the accumulator, we put a number x into position A[$2e$], if e is the (biased) exponent of the number and the last mantissa bit is 0. If the last mantissa bit is 1, then we put the number into position A[$2e + 1$]. Thus, we "know" the last bit of the mantissa by position in the table. If the entry in A is zero, nothing else happens. If the entry is not zero, then we add to x the value at the proper position of A, zero out this entry, and repeat with the result at the new position determined from the exponent and the last mantissa bit. By separating numbers with a trailing 1 from numbers with a trailing zero in the mantissa, there is no loss of information in this addition.

We can think of the accumulator A as a large staggered sum. When the current value is extracted as a floating-point number, the summands are added up in a loop. Summation begins with the highest two nonzero entries. They are added with a side calculation determining the precise error in the addition. The error is re-entered, done much as the addition before, each time increasing the gap between the highest entry and the second highest. This diminishes the influence the second largest entry has on the result, until the gap exceeds the mantissa length and the "low-order" bits have no influence at all on the result. At this point, the accurately rounded result is evident.

This implementation is fairly fast on accumulating values, but takes extra time to extract current values. In some situations it is more efficient than the first implementation, but not always.

3.1.5 Segment Intersection Summary

The segment intersection techniques presented do not concern the global strategy that is used when many segments must be intersected. Instead, they only concern the computation on two segments, a computation that one would like to implement in a utility package and forget from then on. As we see, however, the intermediate precision needed to get reliable answers is unexpectedly high: roughly double for testing whether there is an intersection, and roughly triple to compute the actual intersection coordinates. Textbook algorithms on segment intersection do require testing the equality of intersection point coordinates; e.g.,deBerg et al. [6], Chapter 2. but assume that the test is trivial. As we have seen, that is a poor assumption.

3.2 Polyhedral Intersection

So far, we have not addressed proliferation and irrationality. We discuss those subjects in the context of polyhedral intersection. The intersection of two polyhedra is an excellent test case, because it is a complex algorithm that relies heavily on making incidence decisions correctly. It is also an ideal subject to discuss proliferation, because the intersection is again a polyhedron, as well as irrationality, which arises from rotations. Finally, a well-structured polyhedral intersection algorithm can be reasonably easily extended to an intersection algorithm for curved solids.

3.2.1 Sugihara's Method

Sugihara and Iri [7] proposed an exact polyhedral intersection algorithm that accounts for all sources of robustness problems. The geometric elements are represented without redundancy, giving only the coefficients of the plane equations of the faces. Vertex coordinates are computed on demand in the manner of Section 3.1.3. That is, with plane equations of the form $ax + by + cz + d$, define

$$
\begin{aligned}
D_x &= -\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix} \\
D_y &= -\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix} \\
D_z &= -\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix} \\
D_w &= \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}
\end{aligned}
$$

Then the coordinates of the intersection of those three planes are

$$(D_x/D_w,\ D_y/D_w,\ D_z/D_w)$$

where we assume that $D_w \neq 0$, that is, that the plane normals are linearly independent.

It is important to realize that the representable polyhedra are *not* restricted to convex polyhedra only. This is because the vertex coordinates are computed when needed, as intersection of adjacent face planes. Thus, edges can be represented topologically, by the names of the incident vertices, and edge loops delimiting a face can be defined purely topologically without any need for explicit coordinate information. All coordinates can be derived from the face plane equations as needed.

If we assume that the coefficients are B-bit integers, then roughly $3B$ bits are needed for vertex coordinates without division. It is, however, better to use coefficients d that have $2B$ bits, as this leads to a more uniform grid of representable planes. In that case, we need vertex coordinates that are roughly quadruple precision.

To test vertex/plane incidence accurately, we evaluate the plane equation with the point coordinates exactly, for which quintuple precision is necessary.[4] Incidence means that the determinant vanishes:

$$\begin{vmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{vmatrix} = 0$$

Note that we do not need to compute vertex coordinates explicitly for this test.

A key observation settles concerns about digit proliferation. Although we do require quintuple precisions to make reliable answers in the course of the computation, the output polyhedron inherits all surfaces from the two input polyhedra. Therefore, the output polyhedron does not require a higher precision than the input polyhedra, in order to be represented. That is, no digit proliferation takes place. This is in contrast to segment intersection when the intersection points should be used to define new segments.

Is the choice to represent only plane equations the key factor that eliminates the growth of precision? Since the inheritance of the input planes by the output polyhedron is a geometric fact, the answer must be no. Even though it would appear that with a vertex-based polyhedral representation there is an exponential precision growth, as is the case for segment intersection, the five-fold increase in precision happens only once and never thereafter.

[4] To be precise, a few additional bits are needed to account for the summation.

3.2.2 Representation under Rotation

We have seen that the intersection of two polyhedra can be done with exact arithmetic completely correctly if the input polyhedra are correct. But are they? To answer this question we need to consider where input polyhedra come from. In particular, there is a problem with rigid body rotations.

The unit square is eminently representable, with minimal precision requirements. When rotated by 45° about the origin, however, it no longer is: Neither all line coefficients nor all vertex coordinates are rational numbers after the rotation. Therefore, infinite precision would be needed. The implication is that when a proper, representable polyhedron is rotated by an angle, then it no longer needs to be representable. One expects that a *rounding* operation should be used. But as demonstrated by Sugihara, naive rounding can change the polyhedron into an incorrect one with self-intersections. This happens when there are very small features.

We could imagine using rotations by angles that preserve the rationality of coefficients and coordinates. Such rotations correspond to the rational points on the unit circle which are obtained with rational values for t from

$$\begin{aligned} x(t) &= \frac{1-t^2}{1+t^2} \\ y(t) &= \frac{2t}{1+t^2} \end{aligned}$$

Note that rational rotations do not form a group. That is, the composition of several rational rotations need not be representable as a single rational rotation.

Sugihara and Iri avoid the problem as follows. To guarantee a legal polyhedron, we need to guarantee that the rounding operation after a rotation preserves correctness. This is trivial for trihedral polyhedra that have no small features, such as cuboids that are larger than a minimum size. When a complex polyhedron is represented in dual form, by recording both the boundary structure needed to do Boolean operations, and the CSG tree by which the polyhedron was constructed, then we may apply the rotation to the trihedral primitives and reconstruct the rotated polyhedron from the CSG representation. Although not cheap, doing so guarantees that all polyhedra are correct, a requirement to have a fully robust intersection algorithm with exact arithmetic.

3.2.3 Fortune's Method

A different strategy for polyhedral intersection has been proposed by Fortune [8]. The idea is to avoid exact computation when floating-point computation is sufficiently accurate.

Again, the primary geometry representation is the implicit plane equation $ax + by + cz + d = 0$, where the coefficients are B-bit integers except for d whose precision is $2B$. In contrast to Sugihara and Iri, Fortune tolerates self-intersection in polyhedra as the result of a rigid-body motion, but subsequently extracts the *core*, always a legal polyhedron. Figures 5–7 illustrate the concept.

Call a numerical primitive computation a *predicate* if its value impacts the flow of control of an algorithm, and a *constructor* if its value is used to define geometric data. The evaluation of predicates usually only requires sign determination but must be correct. The evaluation of constructors may tolerate some error in view of the rounding scheme that extracts the core.

Both predicates and constructors are integer polynomials. A preprocessor provides an implementation of the evaluation of those polynomials; Fortune and Van Wyk [9]. By considering the degree and form of the polynomial, as well as the precision of its coefficients, the preprocessor determines whether a floating-point evaluation is sufficiently accurate to evaluate the polynomial without uncertainty. Based on the determination, the preprocessor generates a C++ code fragment that implements the evaluation.

At the time, Fortune's experiments indicated that the running time increased by about a third due to the times that exact arithmetic was used, in less than 10% of the cases. We conclude that exact evaluations are costly, but that they are needed only some of the time. Thus, an implementation that uses exact evaluation only when needed has a clear advantage over an implementation that always evaluates exactly.

3.3 Curved Geometric Primitives

In the case of planes and polyhedra, we saw that quintuple precision is needed to evaluate incidence exactly. Already for quadratic surfaces, exact point/surface tests become daunting. Consider the quadratic surface

$$ax^2 + by^2 + cz^2 + dxy + eyz + fzx + gx + hy + iz + j = 0$$

where the coefficients are integers and

$$-L \leq a, b, c, d, e, f \leq L$$
$$-L^2 \leq g, h, i \leq L^2$$
$$-9L^3 \leq j \leq 9L^3$$

Using Sturm sequences, Yu derives in his thesis [10] an estimate of L^{720} for the precision required to separate two intersections of quadric surfaces. This clearly eliminates a straightforward, exact-arithmetic strategy for curved geometric primitives from further consideration.

The pessimistic bound reflects of course the manner in which the computation separates near, but unequal, points accurately. There is reason to believe that, as in the linear case, certain applications permit much more efficient exact methods.

4 Symbolic Reasoning

A key issue in geometric computations is to achieve consistent evaluation of predicates and constructors as stated in Section 3.2.3. It has been observed that the predicates answer geometric questions. Therefore, the problem of recognizing that such a decision is implied by earlier decisions already made lies in the domain of geometric reasoning. It is held that we can tolerate an incorrect decision in a borderline case as long as it is consistent with all other decisions. An example of that concept has been developed fully for Voronoi diagram computations by Sugihara [11]. The algorithm may well produce an incorrect diagram but it always computes correct topological data structures and will never fail.

There is a rich literature on geometric reasoning, but its tools are very general and have been developed to prove geometric theorems. This suggests that it is better to specialize reasoning to focus only on the predicates and their interaction that arise in a particular geometric application. We illustrate this thought with the case of polyhedral intersection.

4.1 Incidence Decisions in Polyhedral Intersection

As we have seen before, polyhedral intersection ultimately rests on incidence predicates that test whether the intersection of three planes is on a fourth plane, or, more symmetrically, whether four planes meet in a common point. Reconsidering the example of Figure 1, when the two edges are close, we could decide

either intersection or nonintersection. As long as the decision is made consistently, it should lead to consistent data structures. Either choice can be defended on grounds that the input data is probably not accurate as given and that there is no way to know the "right" decision.[5]

To make consistent incidence decisions requires posting an incidence choice to all affected other intersections. Thus, when we decide, on the front face of the cube in Figure 1, that the front edge of the tetrahedron intersects the edge of the cube, then we should make that decision known to the top face as well where the same question is asked for a second time. The difficulty is that we have no control over the sequence in which the faces are examined, so that incidence in the front face may well be investigated *after* it has been decided in the negative on the top face. So, rather than posting nonincidence, which degrades performance because of its volume, we need a way to undo a nonincidence decision previously made.

Another consideration is to minimize the occurrence of equivalent predicates and to reduce all decisions to as few a set of predicates as possible. Doing so reduces opportunities for unrecognized inconsistencies. Therefore, in the following list of different types of incidence decisions, we reduce some of them to a set of *primary* decisions. The following two decisions are considered primary and are computed from the geometric data.

1. $v \cap f$: The vertex v must be sufficiently near the face plane of f, an ϵ-judgement.

2. $e \cap e'$: e and e' must be sufficiently close. Intersect e with one face plane adjacent to e' and ascertain that the intersection is on all other face planes adjacent to e'.

The expression preceding the colon is the one we test for, the description following describes how we implement the predicate and why. If we restrict, furthermore, the geometric data to face plane equations only, then the implementation of the two primary predicates consists in each case of one or more tests whether four planes meet in a common point.

In addition, we have the following secondary predicates whose implementation is reduced to the primary ones. Again, the expression describes the condition we test for, and the text explains what to do and why.

3. $v \cap e$: Let f be a face adjacent to e. The vertex v must be incident to the face plane of each adjacent face f.

[5]Note, however, that on pragmatic grounds one could prefer decisions that do not lead to very small structures in the result.

4. $v \cap v'$: The vertex v must be incident to each edge incident to the vertex v'. That is, v must be on every face plane incident to v' and, vice versa, v' must be on every face plane incident to v.

5. e on f: The vertices on the edge must be incident to the face plane of f.

6. e collinear with e': Let f be a face adjacent to e. The vertices on e' must be incident to the face plane of each adjacent face f. Conversely, the vertices on e must be on every face plane f' incident to e'.

7. $e \cap f$: Test if the edge intersects the face plane transversally, i.e., whether the vertices of e are on opposite sides of f, or one of them is on f but the other is not.

8. $f \cap f'$:
   Every vertex in f must be incident to the face plane of f', and every vertex in f' must be incident to the face plane of f.

Note the difference between cases (7) and (8).

We see now that there is a need to iterate incidence tests because, as vertices are placed into faces and face planes, the incidence decision of the higher-dimensional entities are affected and should be re-examined. Also, to establish symmetry, some tests require multiple ϵ-judgements. Every incidence test benefits from working with the primary geometric data.

There is evidence in practice that a carefully engineered implementation observing these incidence definitions achieves a good measure of consistency. However, there is no proof that the resulting implementation is correct, and it is widely held that this is not the case. The difficulty, from a reasoning perspective, is that on close proximity it is entirely possible that some, but not all, vertices of a face lie in another face. In this case we must reason about a line in which the two face planes intersect and ascertain that the vertices above the other face plane are all on one side of the line, and consistently for both planes. Worse, we now are obligated to ensure also that all vertices, deemed to lie in the other plane, lie on this intersection line as well.

Considerations of this kind make it clear that it is not a trivial matter to prove that an algorithm based on the reasoning approach is indeed fully cognizant of all possibilities and therefore robust. There are examples where the reasoning about "simple" point/line configurations appears to be equivalent to proving geometry theorems; Hoffmann [3], Section 4.4.1. This indicates that, as the exact approach,

there is a complexity barrier preventing us from relying solely on a reasoning approach in geometric computation. However, Hopcroft and Kahn [12] prove that it is possible to consistently intersect a convex polyhedron with a planar half space.

A central difficulty for the reasoning approach is that deductions about geometric incidences are based on a notion of "nearness" or "epsilontics." As such, they must follow unfamiliar logical rules. For example, let us fix a threshold distance below which we judge that points are coincident. We may find that point A is coincident with point B and point B coincident with point C, but that point A is not coincident with point C. See also Figure 8.

Another example of the strangeness of reasoning about nearness is shown in Figure 9. Here the vertex v is determined by computation to lie on both adjacent face planes, yet it cannot be placed on the common edge because the distance, in each face plane, is too great. Such a situation might be resolved by arguing that there should probably an object simplification that eliminates such small, slender features from the input polyhedra.

Finally, we stress that when the algorithms are based on floating-point arithmetic, it is advisable to use, where possible, numerically stable and precise algorithms developed by the numerical analysis community. For instance, determining the coordinates of the intersection of three planes from the determinants of Section 3.2.1 is less precise than using, say, the QR method; e.g., Golub and van Loan [13], Section 7.5.

5 Interval Computation

We pretend to compute with real numbers in the design of geometric algorithms. In reality, however, we compute with finite approximations of numbers. In the case of integer arithmetic, the approximation is exact and the only problem we have to face is overflow. Rational arithmetic is also exact because it is derived from exact integer arithmetic. As we have seen, rational arithmetic is an unsatisfactory solution in the nonlinear case, because the needed precision is too large and some results are not rational. In the linear case rational arithmetic does better, especially when, due to specific properties of the application, simplifications ensue that allow us to get away with limited precision.

Floating-point arithmetic creates an illusion of real arithmetic. By nature, floating-point numbers are peculiar rational numbers with a peculiar range of representability, as illustrated in Figure 4. How can we work with real numbers, and

do so economically?

Symbolic algebraic computation systems work with real algebraic numbers.[6] An algebraic number r is represented by a polynomial (whose root r is) and by a separating interval (a pair of rational numbers of arbitrary precision) that contains r and no other root of the polynomial. This very general approach cannot deliver the performance needed for geometric computations, but the interval idea can be adapted to deliver a useful tool — at least that is the expectation. Instead of rational interval bounds we will discuss the use of floating-point intervals. A floating-point interval is denoted

$$[x] = [\underline{x}, \overline{x}]$$

where $\underline{x}$ and $\overline{x}$ are floating-point numbers and $\underline{x} \leq \overline{x}$. The interval $[x]$ represents all real numbers r between the interval bounds:

$$[x] = \{r \in \mathbf{R} \mid \underline{x} \leq r \leq \overline{x}\}$$

Thus, we may compute with real numbers using precisely defined approximations and automatically retain the knowledge of how closely we have delimited the value of a real number r that would be the true result of a computation with real numbers. We will use the notation of Hammer et al. [14]. The key objective is to come up with an interval enclosure that is as tight as possible.

5.1 Interval Basics

We begin with a few elementary definitions for the real intervals $[x]$ where the interval bounds, $\underline{x}$ and $\overline{x}$, are real numbers. All arithmetic operations are assumed to be exact at first. We explain later what changes are necessary when implementing these concepts and operations using floating-point numbers and operations. We stress that the implementation has to make sure that the machine arithmetic used does not violate the enclosure property. Only then do we have a solid foundation on which to build.

5.1.1 Real Intervals

The (absolute) *diameter* of the interval $[x]$ is $d([x])$, the *radius* is $r([x])$, and the *midpoint* is $m([x])$, where

$$d([x]) = \overline{x} - \underline{x} \qquad r([x]) = \frac{\overline{x} - \underline{x}}{2} \qquad m([x]) = \frac{\overline{x} + \underline{x}}{2}$$

[6] And with a few special transcendental numbers

The *smallest* and *greatest absolute value* of an interval is

$$\begin{aligned} \langle [x] \rangle &= \min\{|x| \mid x \in [x]\} \\ |[x]| &= \max\{|x| \mid x \in [x]\} = \max(|\overline{x}|, |\underline{x}|) \end{aligned}$$

Note that $0 \in [x]$ iff $\langle [x] \rangle = 0$. The *relative diameter* tells us the quality of an approximation by interval. It is defined by

$$d_{\text{rel}} = \begin{cases} \dfrac{d([x])}{\langle [x] \rangle} & \text{if } 0 \notin [x] \\ d([x]) & \text{otherwise} \end{cases}$$

Basic arithmetic with interval is as follows:

$$\begin{aligned} [x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\ [x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\ [x] \cdot [y] &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \ \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \end{aligned}$$

Division requires that the interval by which to divide does not contain zero.

When computing functions of real variables, we can extend the function to interval arguments under certain circumstances. Specifically, let $\phi : D \subseteq \mathbf{R} \to \mathbf{R}$ be a real-valued function that is continuous on every closed interval in its domain D. We extend ϕ to interval arguments by

$$\phi([x]) = \{\phi(x) \mid x \in [x]\}$$

The extension is called the *interval extension* of ϕ. Because of continuity, $\phi([x])$ is again an interval. In particular, the following extensions are of interest:

$$\begin{aligned} [x]^2 &= [\langle [x] \rangle^2, |[x]|^2] \\ \sqrt{[x]} &= [\sqrt{\underline{x}}, \sqrt{\overline{x}}] \qquad\qquad 0 \leq \underline{x} \\ e^{[x]} &= [e^{\underline{x}}, e^{\overline{x}}] \\ \log([x]) &= [\log(\underline{x}), \log(\overline{x})] \end{aligned}$$

The real power function x^n extends as

$$[x]^n = \begin{cases} [\underline{x}^n, \overline{x}^n] & \text{if } 0 < \underline{x} \text{ or } n \text{ odd} \\ [0, |[x]|^n] & \text{if } 0 \in [x] \text{ and } n \text{ even} \\ [\overline{x}^n, \underline{x}^n] & \text{if } \overline{x} < 0 \text{ and } n \text{ even} \end{cases}$$

The basic problem in interval arithmetic is to compute the range of the interval extension of a function f on an interval $[x]$. A natural approach in the cases where f is defined by an expression or a similar computation would be to substitute $[x]$ for x in the computational steps, that is, to do an *interval evaluation* of f (denoted by $f_{[]}$). For example, if we take a polynomial function

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

the interval evaluation would then be

$$f_{[]}([x]) = [a_0] + [a_1][x] + [a_2][x]^2 + \cdots + [a_n][x]^n$$

where $[a_k]$ (in abuse of the notation) is the *thin* interval $[a_k, a_k]$. We observe the containment

$$f([x]) \subseteq f_{[]}([x]) \tag{1}$$

Usually, the straightforward interval evaluation overestimates the true range of the interval extension. This is a key problem in interval arithmetic: The tightness of an enclosure is a measure of the quality of information we have about the true solution. Loose enclosures provide little information.

A general technique to reduce this overestimation is the *centered form*, derived from the mean-value theorem. Assume that f is differentiable on D, and that $c, x \in [x]$, where c is fixed. Then

$$f(x) \subseteq f(c) + f'([x])([x] - c)$$

The righthand side is the centered form.

5.1.2 Extended Intervals

We extend the interval concept to permit infinite bounds. Intervals with one or both infinite bounds are called *extended intervals*. Elementary operations with extended intervals are as expected. In the case of division, the following cases arise if $0 \in [y]$:

$$[x] \,/\, [y] \;=\; \begin{cases} [-\infty, +\infty] & \text{if } \underline{x} < 0 < \overline{x} \text{ or } [x] = 0 \text{ or } [y] = 0 \\ [\overline{x}/\underline{y}, +\infty] & \text{if } \overline{x} \le 0 \text{ and } \underline{y} < \overline{y} = 0 \\ [-\infty, \overline{x}/\overline{y}] \cup [\overline{x}/\underline{y}, +\infty] & \text{if } \overline{x} \le 0 \text{ and } \underline{y} < 0 < \overline{y} \\ [-\infty, \overline{x}/\overline{y}] & \text{if } \overline{x} \le 0 \text{ and } 0 = \underline{y} < \overline{y} \\ [-\infty, \underline{x}/\underline{y}] & \text{if } 0 \le \underline{x} \text{ and } \underline{y} < \overline{y} = 0 \\ [-\infty, \underline{x}/\underline{y}] \cup [\underline{x}/\overline{y}, +\infty] & \text{if } 0 \le \underline{x} \text{ and } \underline{y} < 0 < \overline{y} \\ [\underline{x}/\overline{y}, +\infty] & \text{if } 0 \le \underline{x} \text{ and } 0 = \underline{y} < \overline{y} \end{cases}$$

The first case in the division is due to the fact that we have no information on the quotient $0/0$, hence there is no information on the result.

In the special case that x is a thin interval, i.e., the lower and upper bound are the same, and $[y]$ has at least one infinite bound, we have

$$x - [y] = \begin{cases} [-\infty, +\infty] & \text{if } [y] = [-\infty, +\infty] \\ [-\infty, x - \underline{y}] & \text{if } [y] = [\underline{y}, +\infty] \\ [x - \overline{y}, +\infty] & \text{if } [y] = [-\infty, \overline{y}] \end{cases}$$

5.1.3 Floating Point Arithmetic

We implement interval computations with floating-point numbers. To maintain the strict enclosure of real results, however, we must make sure that the arithmetic operations on the floating-point numbers are rounded properly.

Rounding up of the real number r means the nearest machine-representable floating-point number $\triangle(r) \geq r$ that is not smaller than r. If r can be represented exactly, then of course $\triangle(r) = r$. Similarly, *rounding down* means the nearest machine-representable floating-point number $\bigtriangledown(r) \leq r$ that is not greater than r.

It is crucial that the arithmetic operations with floating-point intervals round up or down such that the resulting interval is guaranteed to include all real numbers that would be the result with exact arithmetic. For the lower bound computations this means rounding down, for the upper bound computations it means rounding up. Let $+_\triangle$ and $+_\bigtriangledown$ denote machine addition that rounds the result up or down, and similarly $-_\triangle$ and $-_\bigtriangledown$ for subtraction. Then we have to implement, for example,

$$\begin{aligned} [x] + [y] &= [\underline{x} +_\bigtriangledown \underline{y}, \overline{x} +_\triangle \overline{y}] \\ [x] - [y] &= [\underline{x} -_\bigtriangledown \overline{y}, \overline{x} -_\triangle \underline{y}] \end{aligned}$$

The standard mode of rounding floating-point numbers is to round the magnitude of the number. This is useless for interval implementation because employing such operations invalidates the enclosure property and re-introduces uncertainty into the operations. Many floating-point units (FPU) allow mode change to the appropriate rounding type. This necessitates some assembler code so directing the FPU. One should be careful that the optimizer does not remove those directives gratuitously.

We must also account for the fact the the decimal representation of floating-point numbers, in the program input and output, is inexact. Where necessary, this may mean the I/O of hexadecimal numbers, or equivalent techniques that permit precise control over the resulting representation.

5.2 Univariate Root Finding

Univariate root finding comes up a lot in geometric computations, and is considered in principle a solved problem. The solver by Jenkins and Traub [15] is often cited as a comprehensive solution. Many other techniques are available as well, and Press et al. [16] explains several of them. The more general and reliable the root finder, the more complex and delicate the algorithm. The great complexity of the comprehensive methods is due to the intrinsic capriciousness of floating-point arithmetic.

For many practitioners, Newton iteration is considered a good stand-by that is not very complicated and, when started up carefully, usually finds one of the roots. In some situations Newton iteration is inapplicable. In particular, as everyone knows, multiple roots defeat Newton iteration. Another issue is that Newton iteration finds only one root x^*, and repeated division of the polynomial by $(x-x^*)$ accumulates imprecise coefficients.

In fact, this need not be, and the univariate Newton algorithm is an example that showcases the potential that interval computation can have in conjunction with the exact inner product. We explain first how to evaluate polynomials accurately, and then describe an interval Newton algorithm.

5.2.1 Polynomial Evaluation

The Horner scheme is known to be an efficient and stable evaluation procedure for polynomials with floating-point arguments. The polynomial

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

is evaluated by a loop in the order:

$$(\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

Evaluation of a polynomial near a higher order root exhibits low-order mantissa bit errors that, when taken literally, would yield a multiplicity of sign changes near the actual root. It is therefore not reliably possible to narrow the interval enclosing a root using evaluation in its neighborhood, nor could we so isolate several roots with small separation.

As an example, consider the evaluation of $f(x) = x^4 - 8x^3 + 24x^2 - 32x + 16 = (x-2)^4$ near the quadruple root 2. 61 evaluations are conducted, both using the Horner scheme and using an interval-based polynomial evaluation described in

Hammer et al. [14], Section 4.2. The latter uses the exact inner product in a matrix iteration. The steps are as small as possible in the representation, determined as shown in Table 2.

At the root $r = 2$, the next representable double r' is computed giving the separation from above, $\delta = r' - r$. Define $a = r - n\delta$ and $b = r + 2n\delta$, where n is a user input. Then evaluate the polynomial from a to b, stepping through all representable floating-point numbers in-between, and observe the sign of the computed value. In the case of the quartic polynomial, 40 evaluations are made on either side of the (4-fold) root 2, from $a =$ 0x3fffffff fffffd8 to $b =$ 0x40000000 00000028. In the case of the cubic polynomial $(1-x)^3$, there are also 40 evaluations on either side of the (triple) root 1, from $a =$ 0x3feffff fffffd8 to $b =$ 0x3ff00000 00000028. The results for $n = 20$ are shown in Table 2. Note the many sign reversals suffered by the Horner evaluation.

It is apparent that the Horner evaluation is not able to evaluate floating-point arguments in the vicinity of multiple roots, even for low-degree polynomials as the ones used here. Note, however, that the exact inner product plays an critical role in the interval evaluation because it allows accurate evaluations for the matrix iteration.

5.2.2 Interval Newton

We consider $f(x) = 0$ where f is continuously differentiable over the reals. This is true for univariate polynomials. By the mean value theorem,

$$f(m([x])) - f(x^*) = f'(\xi) \cdot (m([x]) - x^*) \qquad x^*, \xi \in [x]$$

We assume that x^* is a zero of $f(x)$. Define

$$N([x]) = m([x]) - \frac{f(m([x]))}{f'([x])}$$

Then

$$x^* = m([x]) - \frac{f(m([x]))}{f'(\xi)} \ \in \ N([x]) \tag{2}$$

This means that the zero x^* is both in $[x]$ and in $N([x])$ and therefore in the intersection of these two intervals. Beginning with a starting interval $[x]^{(0)}$, we can iterate

$$[x]^{(k+1)} \ = \ [x]^{(k)} \cap N([x]^{(k)}), \qquad 0 \notin f'([x]^{(k)}) \ \text{ and } \ k = 0, 1, 2, ... \tag{3}$$

Since all intervals are contained in the initial interval $[x]^{(0)}$ the method cannot diverge. This is in contrast to traditional Newton iteration that suffers from this problem near a multiple root.

The geometric interpretation of an iteration step is closely analogous to the geometric interpretation of the classical Newton iteration, and is shown in Figure 10. We assume that $f'([x]^{(k)}) = [\underline{g}, \overline{g}]$ and that the lines with the bounding slopes intersect the x-axis at l and r, respectively. That is, $N([x]^{(k)}) = [l, r]$. If the intersection in a Newton step is empty, then we know for certain that there is no root of f in the interval $[x]^{(k)}$.

Using extended interval arithmetic, we can dispense with the restriction that $0 \notin f'([x]^{(k)})$. If the interval $f'([x]^{(k)})$ contains zero, we obtain extended intervals for $N([x]^{(k)})$, namely

$$N([x]^{(k)}) = [-\infty, r] \cup [l, \infty]$$

Intersection with $[x]^{(k)}$ results in up to two regular (finite) intervals

$$[x]^{(k+1)} \;=\; [\underline{x}^{(k)}, r] \cup [l, \overline{x}^{(k)}] \tag{4}$$

This is illustrated in Figure 11. In summary, the following is true:

> Let $f : D \subseteq \mathbf{R} \to \mathbf{R}$ be a continuously differentiable function, and let $[x] \subseteq D$ be an interval in the domain of f. Then
>
> $$N([x]) = m([x]) - \frac{f(m([x]))}{f'([x])}$$
>
> has the following properties:
>
> 1. Every zero $x^* \in [x]$ of f satisfies $x^* \in N([x])$.
> 2. If $N([x]) \cap [x] = \emptyset$, then there is no zero of f in $[x]$.
> 3. If $N([x])$ is contained in the interior of $[x]$, then there exists a unique zero of f in $[x]$ and hence in $N([x])$.

For a proof see, e.g., Neumaier [17]. Based on these facts, we can implement an interval Newton solver that reliably finds the roots of polynomials. While the solver does not miss root enclosures, it may fail to sufficiently refine computed intervals so that isolated roots can be separated. Moreover, there appears to be no reliable way to determine the multiplicity of roots using the interval Newton algorithm.

As an example, we find roots for the polynomial

$$\begin{aligned}(x-1)(x-2)^2(x-3)^3(x-4)^4 = \\ 27648 - 110592x + 192384x^2 - 192832x^3 + 123852x^4 \\ - 53428x^5 + 15715x^6 - 3118x^7 + 400x^8 - 30x^9 + x^{10}\end{aligned}$$

All roots are found without trouble.

5.3 Intervals in Geometric Computations

In Section 4 we explained the difficulties devising a logic by which to reason with tolerances. Since intervals are tolerances, computing with them is subject to similar problems, except that, with careful implementation of interval operations and theorems such as the one expressed by Equation 1, we do obtain automatic enclosures of a "true" result. We have to take "true" in a technical sense as meaning *the result of real arithmetic when the input data is exact as written*. For simpler geometric problems, such as segment intersection, the assumption "exact as written" is easy to defend. For complex data structures it poses some questions. For instance, suppose that the input is a polyhedron with polygonal faces. Is it reasonable to assume that the vertices of each face are coplanar when their coordinates are so understood? Fortunately, we can relax the vertex coordinates to be intervals, and the same is true for the coefficients of the plane equations. Nevertheless, there is an issue that has to be accounted for.

Assuming we have satisfactorily established such assumptions, the main work now focuses on the following:

1. Devise an efficient evaluation of the geometric problem.

2. Determine a proper course for interpreting enclosures that cannot be refined, or results that cannot be guaranteed.

In the case of the univariate interval Newton algorithm, for instance, we may be unable to determine the multiplicity of a root that was successfully enclosed by the computation. Clearly, a key question regarding the second point is how error propagates geometrically in various constructions.

An example of such an investigation is how to enclose the evaluation of the Bézier curve $C(t)$. Interval Bézier curves were discussed in Sederberg and Farouki [18]. The concept was further developed by Hu et al. [19] with an emphasis on robust evaluation. We explain here the more general approach by Wallner et al.

[20]. We will first look at this problem in a general way, when the control points are known to lie in a convex tolerance region, and then consider the special case when the control points are enclosed in 2-dimensional envelopes.

5.3.1 Convex Sets

Convex sets have been widely studied. We summarize here a few of the properties needed to consider the Bézier curve evaluation with convex sets as control points. A set K is convex if, for any two points in K, the segment connecting those points is contained in K. The plane E is a *support plane* of K if K and E have a point in common and K is contained in one of the (closed) half spaces defined by E, i.e., if K lies on one side of the plane. Some examples are shown in Figure 12 with K an ellipse.

The *support function* s_K is a function that maps each unit vector n to the (unique) plane $s_K(n)$ that is orthogonal to n and is a support plane of K. If we fix a coordinate system and define $s_K(n)$ as the distance of this support plane from the origin of the coordinate system, then the support planes have the equation $x \cdot n = s_K(n)$. When K is understood, we omit the subscript. Evidently, the points in K satisfy

$$p \in K \iff p \cdot n \leq s(n) \quad \forall n$$

We define arithmetic operations with convex sets much the same way in which we defined operations on intervals. Note that intervals are just 1-dimensional convex sets. In particular, for a number t, we define tK as the set of points tx where x is in K. The set $(1-t)K_1+tK_2$ is an *affine combination* of K_1 and K_2; if $0 \leq t \leq 1$, then it is a *convex combination*. The set $K_1 + K_2$ is the *Minkowski sum* of K_1 and K_2.

A support function satisfies the following properties. If $s(n)$ is the support function of K, then $s'(n) = ts(n)$ is the support function of $K' = tK$, in particular $s'(n) = s(-n)$ is the support function of $K' = -K$, understanding point coordinates as position vectors.

If s_1 and s_2 are the support functions of K_1 and of K_2, and if $0 \leq t \leq 1$, then the convex combination $K = (1-t)K_1 + tK_2$ is also convex and has the support function $s = (1-t)s_1 + ts_2$. The key is that the coefficients $(1-t)$ and t are nonnegative. More generally,

$$K = \sum_{t_j>0} t_j K_j + \sum_{t_j<0} t_j K_j = \sum_{t_j>0} t_j K_j + \sum_{t_j<0} (-t_j)(-K_j)$$

so that the support function of such a linear combination is

$$s(n) = \sum_{t_j>0} t_j s_j(n) + \sum_{t_j<0} (-t_j) s_j(-n)$$

The distance between the support planes with normal n and with $-n$ is the *diameter* d_n of the convex set. If K is a convex combination $K = \sum t_j K_j$, then the support function is too, so that the diameters are also, $d_n(K) = \sum t_j d_n(K_j)$. Based on these basic properties, we can analyze the evaluation of many types of parametric curves and surfaces.

5.3.2 Bézier Curves

Recall the definition of a Bézier curve; e.g., Farin [21].

$$C(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$

where the P_i are the control points and the $B_i^n(t)$ are the (degree n) Bernstein-Bézier basis functions. The curve points are usually evaluated using the DeCasteljau algorithm, illustrated in Figure 13.

The sign of the ordinary Bernstein basis function is given by $sgn(B_i^n(t)) = sgn(t)^{n-i} sgn(1-t)^i$. Therefore, the support of the convex body $K = \sum B_i^n(t) K_i$ is $s(n) = \sum B_i^n(t) s_i^*(n)$, where

$$s_i^*(n) \;=\; \begin{cases} s_i(n) & \text{if} \quad t<0, n-i \text{ even, or} \\ & \quad\; t>1, i \text{ even, or} \\ & \quad\; 0 \le t \le 1 \\ -s_i(-n) & \text{if} \quad t<0, n-i \text{ odd, or} \\ & \quad\; t>1, i \text{ odd, or} \end{cases}$$

Based on this information, we can evaluate a toleranced function value given toleranced control point values using the basis-function form of the curve definition.

Figure 15 shows an evaluation of a Bézier curve with circular tolerance regions, and the resulting tolerance for the curve point evaluated in this manner. The grey stripe is the resulting "toleranced" curve. Using the DeCasteljau algorithm, note that the tolerance zone bounds are defined by the inner and outer tangents, as shown in Figure 14.

When the parametric domain is extended, as shown in Figure 16, then the error increases substantially outside the standard range $0 \le t \le 1$. This is consistent with well-known analytical results that the accuracy when evaluating in the

Bernstein-Bézier basis is greatest in the standard range; e.g., Farouki and Rajan [22].

5.3.3 Interval Control Points

As a special case, we assume that the coordinates of the control points are intervals. The tolerance regions for control points are thus rectangles. Since rectangles are convex, the results of the previous section hold, justifying the interval extension evaluation of individual curve points. Again, as before with disks, the tolerance region of the value is governed by the inner and outer tangents, as shown in Figure 17.

5.3.4 Semantics

Do these constructions deliver tight bounds? Yes, but we have to be precise in what we mean: Given a convex set K as control point p, we interpret this to mean that each Euclidean point in K may be chosen as control point. Thus the "toleranced" curve is the union of exact curves obtained from all possible choices of control points in the given sets. On geometric grounds, therefore, a DeCasteljau point of such an exact instance curve belongs to the toleranced curve. By the preceding analysis this means that the exact evaluation of the tolerance zones, by the DeCasteljau algorithm, must deliver a correct set of curve points for the particular (exact) parameter value t.

Note, however, that there is no assurance that a computer implementation can carry out an exact evaluation. Rather, a correct interval implementation of the method, as sketched before, will deliver a guaranteed enclosure. Depending on the degree of the curve and the value t, the computed enclosure may be not as tight as representationally possible.

6 Conclusions

Exact arithmetic offers significant advantages: it avoids uncertainty of incidences, intersections, and so on, and it allows reducing special cases that arise otherwise from the looseness with which predicates and constructors are evaluated in floating-point arithmetic. For the mathematical-minded, the preciseness and certainty is probably a great re-assurance, but in the practical setting the cost of carrying out exact arithmetic becomes an issue almost immediately. A simple rule

of thumb here would be that in the piecewise linear domain, the world of line segments, polygons and polyhedra, the cost of exact arithmetic can be defended in mission-critical applications. Exact arithmetic in the nonlinear domain is much more costly and is not often afforded in practice. The paper by Keyser et al. [23] describes an exact implementation of nonlinear surfaces.

An issue less prominently discussed when using exact arithmetic is the considerable difficulty of getting the exact input data to be correct: When giving the vertex coordinates of a polygon for example, coplanarity becomes an issue for anything but triangles. When giving the coefficients of the support planes for polygons joining at a polyhedral vertex, it is not automatic that all planes meet in the same point when the vertex is not trihedral. Those seemingly innocent issues will impact the correctness of the exact computation.

Another issue that must be addressed for exact arithmetic is the precision proliferation and the possibility of transcendental coordinates. If the intersections computed from a set of segments become, in turn, end points of segments to be intersected, iteration of this process drives an exponential growth of precision. When a polygon or a polyhedron is rotated by an angle, irrational coordinates can ensue that either ruin the exactness of the representation, or else sharply drive up the cost with additional constructs such as polynomial or transcendental function manipulation.

Symbolic reasoning seeks to address the robustness problem by trying to make sense of imprecise data and finding consistent interpretations. The logic of this process, and a reasonable bound on its complexity, become the issues in this approach. Do several incidences imply a collinearity of other points? Perhaps we are re-proving the theorem of Pappus. In certain cases such "theorems" have been tested with random evaluations, as in the Cinderella implementation discussed in Kortenkamp [24]. This brings us back to reasoning with an uncertain evaluation mechanism, a foundation that may have a measure of robustness that depends inversely on the complexity of the geometric structure and construction, but not a foundation that allows us to forget the lower software layers and take them for granted. To-date, the attempts put forth to argue that symbolic reasoning can confer absolute robustness have all met with skepticism in the community, in cases where complex data structures are involved.

Loosely speaking, interval arithmetic provides guaranteed enclosures and so can be thought of as computing with automatic tolerance estimates. In many cases, such enclosures are sufficiently precise so that we can say with certainty whether a predicate evaluates to true or to false. But there are cases where the enclosure is not sufficiently tight, and then we again face the uncertainty of interpreting its

implication. So, the key task is to look at the problem at hand and figure out how to come up with the tightest enclosure possible at acceptable cost.

An interesting variation of the interval approach has been discussed by Agrawal [25] in his PhD thesis. Agrawal adopts an approximate correctness notion to mean that for the given numeric data there is a perturbation, within a tolerance limit, such that the symbolic data with the perturbed numerical values is a valid representation and is a correct result for analogously interpreted input data using exact arithmetic. Thus, input with interval coordinates is processed by interval arithmetic delivering interval-valued output coordinates that are correct in this sense. Instead of analyzing the utility of the approach for a class of geometric problems, as is normally done, Agrawal develops his method purely formally to a programming model and proves correctness. Thus, as long as a geometric computation can be expressed in the programming model (a given because the model is general), Agrawal's method succeeds in generating correct results. However, it is not clear whether the output coordinates are enclosed with sufficient precision to be useful in practice.

Each of these approaches has some intriguing techniques to showcase. Clever setups of polyhedral intersection address the problem of exactness and precision proliferation. Careful observation of the predicates can lead us to safely use floating-point arithmetic without further safeguards. Common-sense reasoning allows us to make algorithms more robust and, in cases such as Voronoi diagrams, fully reliable. Exact inner product evaluation in conjunction with directed rounding can give enclosures for polygon evaluation down to single-bit errors.

To-date, nobody has put forth a general, winning approach that on the one hand confers absolute robustness while, on the other hand, not costing prohibitively. This means that we need to continue to practice a craft instead of an exact science, and that we must pick and choose from the techniques developed so far pragmatically according to the circumstances and the application.

References

1. J. Lakos. (1996) *Large-Scale C++ Software Design*, Addison-Wesley, Reading, MA.

2. C. M. Hoffmann, J. Hopcroft, and M. Karasick. (1988) "Towards implementing robust geometric computations," *Proc. 4^{th} ACM Symp. on Comp. Geometry*, 106–117.

3. C. M. Hoffmann. (1989) *Geometric and Solid Modeling, An Introduction*, Morgan Kaufman, San Mateo, CA.

4. M. Gavrilova and J. G. Rokne. (2000) "Reliable line segment intersection testing," *CAD 32*, 737–746.

5. H. Ratschek and J. Rokne. (1999) "Exact computation of the sign of a finite sum," *Appl. Math. and Comp.*, 99–127.

6. M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. (1997) *Computational Geometry, Algorithms and Applications*, Springer Verlag, New York.

7. K. Sugihara and M. Iri. (1989) "A solid modeling system free from topological inconsistency," *J. of Inf. Proc. 12*, 380–393.

8. S. Fortune. (1995) "Polyhedral Modeling with Exact Arithmetic," *Proc. 3rd Symp. Solid Modeling*, ACM Press, NY, 225–234.

9. S. Fortune and C. Van Wyk. (1993) "Efficient exact arithmetic for computational geometry," *Proc. 9th Symp. Comp. Geometry*, ACM Press, NY, 163–172.

10. J. Yu. (1991) "Exact Arithmetic Solid Modeling," PhD Thesis, CS, Purdue University.

11. K. Sugihara. (1992) "A simple method for avoiding numerical error and degeneracy in Voronoi diagram construction," *IEICE Trans. Fundamentals, E75-A*, 468–477.

12. J. E. Hopcroft and P. J. Kahn. (1992) "A paradigm for robust geometric algorithms," *Algorithmica 7*,339–380.

13. G. Golub and C. van Loan. (1983) *Matrix Computations*, Johns Hopkins University Press.

14. R. Hammer, M. Hocks, U. Kulisch and D. Ratz. (1995) *C++ Toolbox for Verified Computing, Basic Numerical Problems*, Springer Verlag, New York.

15. M. Jenkins and J. Traub. (1970) "A three stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration," *Numer. Math. 14*, 252–263.

16. W. Press, S. Teukolsky, W. Wetterling, B. Flannery. (1992) *Numerical Recipes in C*, 2^{nd} edition, Cambridge Univ. Press.

17. A. Neumaier. (1990) *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, England.

18. T. Sederberg and R. Farouki. (1992) "Approximation by interval Bézier curves," *IEEE CG&A*, 87–95.

19. C-Y. Hu, N. Patrikalakis, X. Ye. (1996) "Robust interval solid modeling," *CAD*, 807–817 and 819–830.

20. J. Wallner, R. Krasauskas, and H. Pottmann. (2000) "Error propagation in geometric computations," *CAD 32*, 631–641.

21. G. Farin. (1992) *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, Boston; third edition.

22. R. Farouki and V. Rajan. (1987) "On the numerical condition of polynomials in Bernstein form," *Comp. Aided Geom. Design 4*, 191–216.

23. J. Keyser, T. Culver, D. Manocha, S. Krishnan. (2000) "Efficient and exact manipulation of algebraic points and curves," *CAD*, 649–662.

24. U. Kortenkamp. (1999) "Foundations of Dynamic Geometry," PhD Thesis, Informatik, Swiss Fed. Inst. of Technology.

25. A. Agrawal. (1995) "A General Approach to the Design of Robust Algorithms for Geometric Modeling," PhD Thesis, Comp. Science, Univ. of Southern California.

List of Tables with Captions

1. ESSA algorithm (Ratschek and Rokne)
2. Polynomial evaluation of representable doubles around multiple root; 40 doubles on either side ($n = 20$).

List of Figures

1. Intersection of a cube with a tetrahedron
2. Possible inconsistent face partitions
3. The intersection of representable segments need no be representable
4. The grid of floating-point numbers
5. Ideal polyhedron.
6. The planes are perturbed towards each other, giving an improper polyhedron
7. The core of the improper polyhedron is extracted
8. “Nearness” that implies coincidence is not transitive
9. Can v be on both faces but not on the connecting edge?
10. Interval Newton step when $0 \in f'([x]^{(k)}$, adapted from Hammer et al. [14], where $c^{(k)} = m([x]^{(k)})$
11. Interval Newton step when $0 \notin f'([x]^{(k)}$, adapted from Hammer et al. [14]
12. Support lines of a convex set
13. DeCasteljau algorithm for a Bézier curve
14. Tolerance zones for linear combinations of disks
15. Bézier curve with toleranced control points
16. Bézier curve with toleranced control points in extended domain
17. Tolerance zones for linear combinations of rectangles

Table 1: **ESSA Algorithm (Ratschek and Rokne)**

E denotes the exponent of a_1, F the exponent of b_1. The terms are stored in two lists of length m and n, one for each sum, and are ordered by decreasing magnitude.

1. *Termination*
   If $m = n = 0$ then $s = 0$; exit.
   If $m > n, n = 0$, then $s > 0$; exit.
   If $n > m, m = 0$, then $s < 0$; exit.
   If $a_1 > n2^F$, then $s > 0$; exit.
   If $b_1 > m2^E$, then $s < 0$; exit.
2. *Initialization*
   Set a', a'', b', b'' all to zero.
3. *Leading Term Analysis*
   If $E = F$, then
   If $a_1 > b_1$, then $a' = a_1 - b_1$,
   else $b' = b_1 - a_1$.
   If $E > F$, then
   If $b_1 = 2^{F-1}$, then set $u = 2^{F-1}$,
   else set $u = 2^F$.
   Set $a' = a_1 - u$, $a'' = u - b_1$.
   If $F > E$, then
   If $a_1 = 2^{E-1}$, then set $u = 2^{E-1}$,
   else set $u = 2^E$.
   Set $b' = b_1 - u$, $b'' = u - a_1$.
4. *Sort*
   Drop a_1 and b_1 from the lists.
   Enter those of the values a', a'', b', b'' that are not zero,
   respectively into the a- and b-lists.
   Re-establish sorting order.
5. *Loop*
   Update m and n;
   Go to Step 1.

Method	negative	zero	positive	sign rev.
$x^4 - 8x^3 + 24x^2 - 32x + 16$				
Horner	15	41	25	29
Interval	0	1	80	0
$-x^3 + 3x^2 - 3x + 1$				
Horner	25	41	15	9
Interval	40	1	40	1

Table 2: Polynomial evaluation on all representable doubles around multiple root; 40 doubles on either side ($n = 20$).

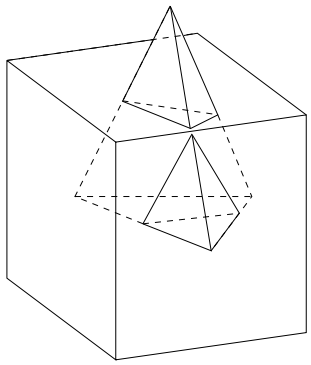

Figure 1: Intersection of a cube with a tetrahedron

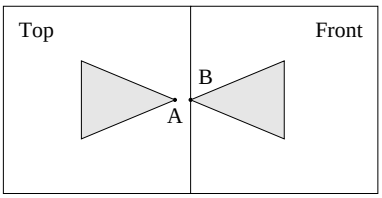

Figure 2: Possible inconsistent face partitions

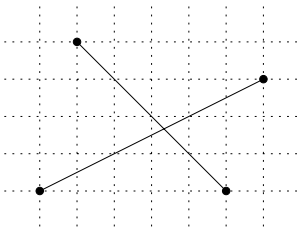

Figure 3: The intersection of representable segments need not be representable

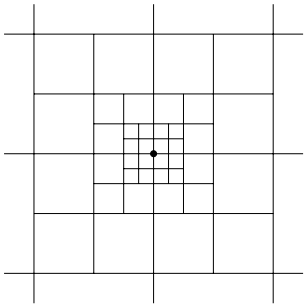

Figure 4: The grid of floating-point numbers

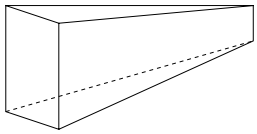

Figure 5: Ideal polyhedron

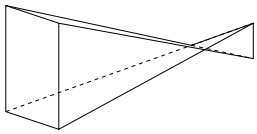

Figure 6: The planes are perturbed towards each other, giving an improper polyhedron

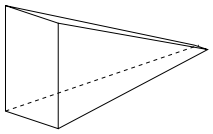

Figure 7: The core of the improper polyhedron is extracted.

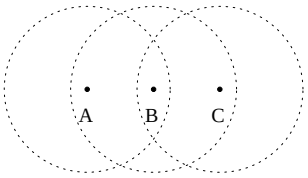

Figure 8: “Nearness” that implies coincidence is not transitive

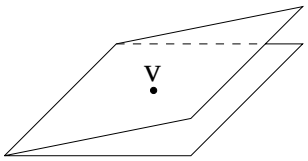

Figure 9: Can v be on both faces but not on the connecting edge?

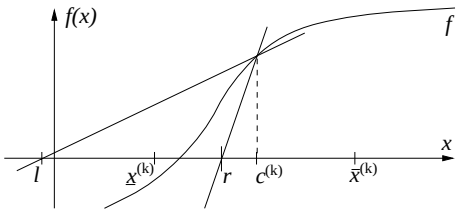

Figure 10: Interval Newton step when $0 \in f'([x]^{(k)}$, adapted from Hammer et al. [14], where $c^{(k)} = m([x]^{(k)})$

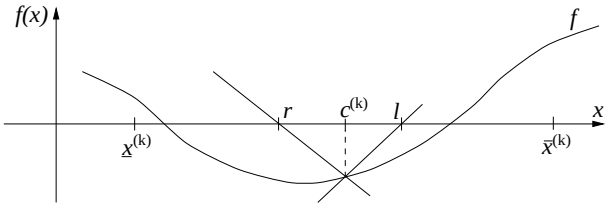

Figure 11: Interval Newton step when $0 \notin f'([x]^{(k)}$, adapted from Hammer et al. [14]

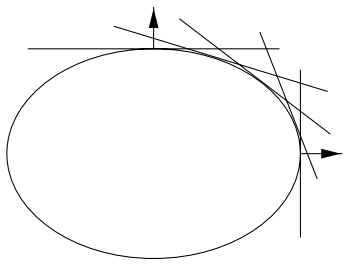

Figure 12: Support lines of a convex set

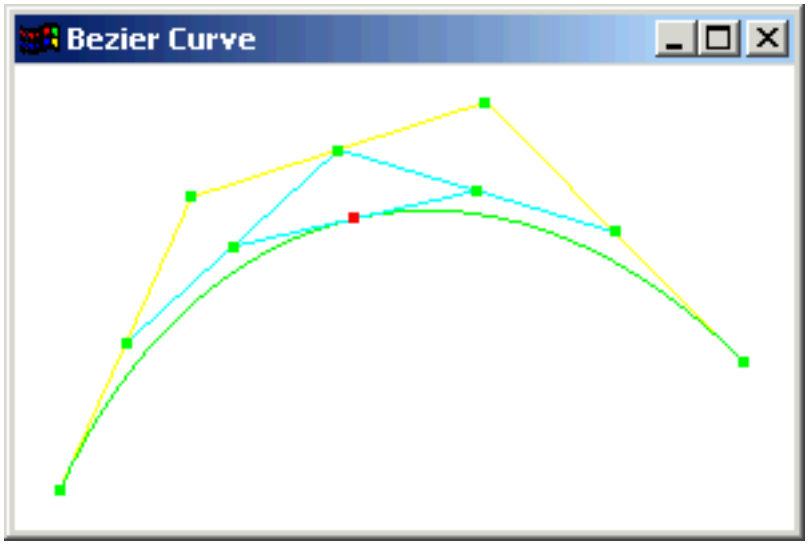

Figure 13: DeCasteljau algorithm for a Bézier curve

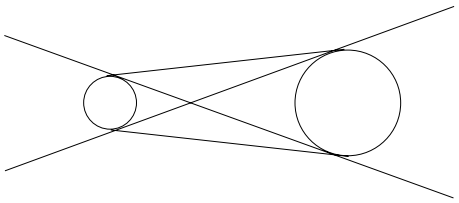

Figure 14: Tolerance zone for linear combinations of disks

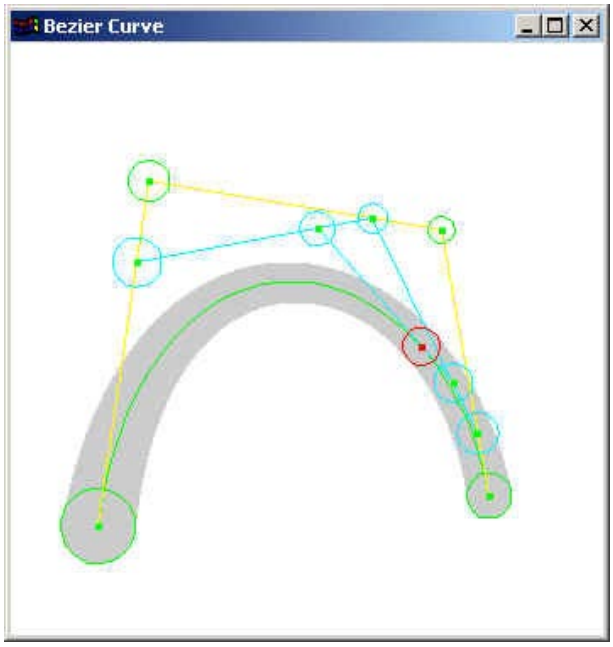

Figure 15: Bézier curve with toleranced control points

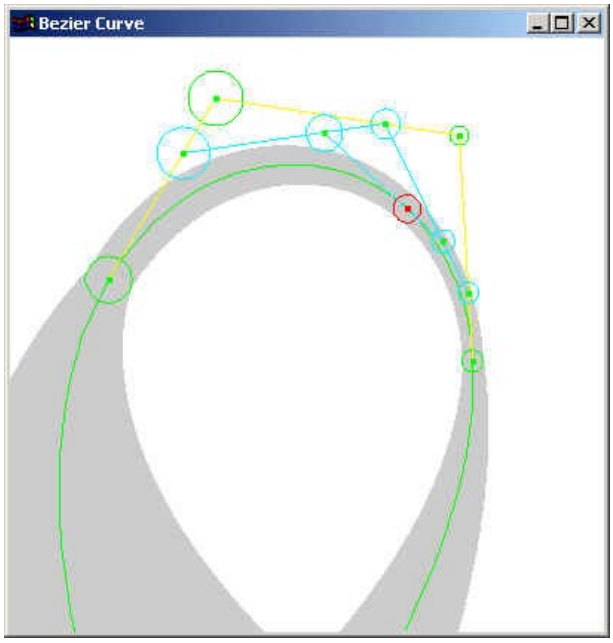

Figure 16: Bézier curve with toleranced control points in extended domain

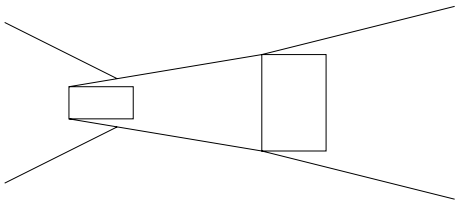

Figure 17: Tolerance zone for linear combinations of rectangles