Robustness Issues in Computational Geometry

John Keyser
Comp 234 Final Paper

1 Introduction

In recent years, the "robustness issue" has gained increasing attention in the field of computational geometry. Some common assumptions in the design of computational geometry algorithms include that that all calculations are performed using exact real arithmetic, and that all objects are placed in general position. However, using exact real arithmetic is not an option available to a programmer, and a programmer will often not be able to ensure that the data given to him is in general position. Thus, practical implementations of the algorithms may fail, even when the underlying algorithm is sound under those assumptions. Such failures are referred to as *robustness problems*, and the attempt to deal with such problems is an area of current interest and work.

In the focus on dealing with robustness, it is important to remember that robustness is only one desirable characteristic of an implementation. Other desirable characteristics include *accuracy* and *efficiency*. Usually, there is a tradeoff among these three aspects, and an implementor can only increase one by decreasing another. For example, several of the robustness techniques which will be described in this survey will result in an approach which is considerably less efficient (i.e. run slower) than a less robust implementation would be. The decision about which approach is "best" will depend on the characteristics of the problem it is being applied to.

There are two main categories of robustness problems. These are problems due to *precision* and problems due to *degeneracies*. In this paper, I will give a brief survey of the issues involved with each of these classes of problems, and discuss some of the solutions that have been proposed for dealing with them. In particular, I will draw on examples from the field of geometric and solid modeling, as an area where many of these problems have been seen and these techniques applied.

There are several possible definitions for what makes an algorithm "robust." Some have even devised systems for specifying the "type" of robustness an algorithm achieves ([DSB92]). A number of approaches which are outlined here claim to be robust. Other approaches only claim to *increase* robustness. Rather than try to examine only approaches which meet a certain robustness criteria, this paper will examine techniques which claim to at least increase robustness in some sense.

The tests performed by various methods are often done for a particular *predicate*. A predicate is just a simple, well defined, test which can be performed on the data and is used to determine the flow of control in a program. It is usually just the sign of some polynomial calculation on input data. An example of a commonly used predicate is the plane orientation test (determining which side of a plane a point lies on), which is usually performed by taking sign of the determinant of a matrix. On the other hand, *constructors* are used to actually generate a new geometric object, and require the *value* of some polynomial calculation on the input data.

Another important consideration in the study of robustness and proposed techniques is the degree of the data being worked with. Most of the classical computational geometry problems have been defined for linear objects (e.g. lines, polygons, polyhedra). Several problems of interest, particularly those in geometric and solid modeling, make use of quadric or higher degree objects. Higher degree objects will, in general, cause robustness problems and execution times to become much worse ([Mil89]).

It should be noted that there have been several other attempts at surveying the robustness issue. Several papers will give a brief survey of robustness issues in general before presenting some new technique. Among those who specifically attempt to survey the robustness issue (to various levels of detail) are: [Hof89b], [Hof89a], [Sch96], [For96], and [Sei94].

2 Precision Problems

The design and description of algorithms can be simplified by the assumption of the *real RAM* model of computation ([For95]). This model assumes that all arithmetic operations are exact, real, and carried out in constant time. An example of how the RAM model can overly simplify analysis is provided in [KS96]. This paper presents an algorithm analyzed by the RAM model as taking only linear time which actually requires $\Theta(n^2 \log n)$ bits of precision in order to accurately represent the output. Because it requires $\Theta(n^2 \log n)$ space, it cannot be linear in time! Computers, being finite machines, are incapable of storing every possible real number. Even for a subset of the reals it is unrealistic to expect that all computations can be exact and performed in constant time.

Obviously, the real RAM model cannot be implemented. Instead, people commonly use floating point arithmetic, which is available on most computers. The problems with using floating point numbers stem from the fact that a constant, finite number of bits are used for all number representations. Given a real number, it can be approximated by a floating point number within some error (which is a function of the number of bits used by the floating point number). In most cases, the floating point approximation is "close enough" and the computation is simple enough that the floating point number can be effectively used as a substitution for the real number. This, coupled with the fact that most modern computers have very fast floating point arithmetic implementations, contributes to the popularity of using floating point. A further justification for the use of floating point numbers comes from the fact that input data are really only approximations, anyway, so it treating such numbers as exact is not "correct."

In some cases, however, the small amounts of error in storing floating point numbers will cause problems. First, two numbers which are actually different might have the same floating point representation. These numbers then appear to be equal when they are not, possibly resulting in incorrect decisions being made in a program. Another problem comes from the growth of error over a series of computations. What starts out as a small, insignificant error might become magnified into a very large error over a series of arithmetic operations.

These basic problems with error accumulation have been well studied in the numerical analysis literature. A number of techniques exist to analyze and contain the growth of such error accumulation for various types of problems. For details see any good numerical analysis book.

Traditional numerical analysis concerns itself with the growth in error where the output is a numeric calculation based on the input. For problems in computational geometry, however, representations usually involve both *numeric* and *combinatoric* data, and choices about combinatoric structure may be made based on numeric data ([HK89]). Thus, error accumulation can lead to incorrect combinatoric information. For example, an inaccurate determination of an intersection point might lead to something with an incorrect topology. Whereas numerical data might be off by a certain amount of error, the combinatorial data is simply right or wrong. While slight inaccuracies in numeric output are often acceptable, usually combinatorial data *must* be correct.

There are a number of approaches that people have taken in attempting to deal with precision problems. These approaches generally fall into two categories. Some approaches continue to use floating point or some other finite precision arithmetic. Another option is to use some form of exact computation.

2.1 Floating Point and Finite-precision

The efficiency and ease of use of floating point arithmetic make it continue to be attractive for numerical computation. In order to achieve robustness, however, special techniques need to be used. Following are descriptions of some of the numerous techniques which have been used to increase robustness while continuing to use finite precision mathematics.

Interval Arithmetic. Interval arithmetic is one method which is used to increase robustness while maintaining the use of floating point arithmetic. With interval arithmetic, each number is stored as a interval. The actual number lies somewhere between the two interval bounds. Computations on the number are actually performed as a set of computations on the interval bounds. For example, $[a, b] + [c, d] = [a + c, b + d]$. Similar rules define the other interval operations. It is important to round the floating point interval bounds correctly in order to ensure that the final interval will contain the answer. Entire books have been written which deal exclusively with the use of interval arithmetic.

By storing the interval bounds as floating point numbers, the interval computations can be performed quickly. There are several drawbacks to using these methods, however. First of all, the computed intervals may be much larger than what is actually necessary. For example, using interval arithmetic to calculate $x(10 - x)$ for $x = [4..6]$ gives:

$10 - x = [10..10] - [4..6] = [4..6]$

$x(10 - x) = [4..6][4..6] = [16..36]$

which is much wider than the actual possible range of [24..25]. For complex computations, the output interval size may grow too large to even be useful. It may be that the starting intervals cannot be made small enough for the final computation to have an acceptably narrow interval size. The intervals themselves may not be able to separate two points which are very close together. Although *incidence transitivity* (where $A = B$ and $B = C$ but $A \neq C$) could be a problem in a naive implementation, it is also possible to eliminate that problem ([HPY96a]). Despite the drawbacks, interval arithmetic has been used in practice, with some success. Hu et el. have even adapted this approach for use on curved surfaces in a solid modeling system ([HPY96b]).

Affine Arithmetic. An extension to interval arithmetic is the use of affine arithmetic ([CS93]). Affine arithmetic attempts to reduce the overly conservative bounds sometimes generated by interval arithmetic. It does so by keeping track of correlations between error introduced at each step in a long computation chain. Each number is stored as an affine form:

$x = x_0 + x_1\varepsilon_1 + ... + x_n\varepsilon_n$

where $\varepsilon_i = [-1..1]$ and represents one source of error. As new potential errors are introduced, new ε_i are introduced. This allows one to accurately predict cancellation of error by reuse of the same variable. For example, the previous calculation of $x(10 - x)$ for $x = [4..6]$ gives:

$x = 5 + 1\varepsilon_1$

$10 - x = 5 - 1\varepsilon_1$

$x(10 - x) = 25 + 5\varepsilon_1 - 5\varepsilon_1 - 1\varepsilon_2 = 25 - 1\varepsilon_2 = [24..26]$

which is much closer to the actual interval, [24..25], than the previous calculation of [16..36] was.

Obviously, the use of affine arithmetic will be slower than the use of "straight" interval arithmetic, but in cases where there might be error correlation from one step of a computation to the next, it can pay off. Affine arithmetic has been used for applications in computer graphics ([CS93]) and in surface intersection problems ([dF96]). Its applicability to more classical computational geometry problems has not yet been looked into.

Epsilon Geometry. Another method closely related to interval arithmetic is the epsilon geometry defined by Guibas et al. ([GSS89]). Epsilon geometry uses "epsilon predicates" to solve problems. These predicates will return intervals which identify regions over which the predicate is definitely true, definitely false, or uncertain. The interval defines a range of amounts by which input data can be perturbed in order to reach the "uncertain" interval (and thus, by implication, the amounts of perturbation which give a true or a false return). Thus, if a predicate returns with a "False" region of points < 0.3 that means that this predicate will always return false as long as the data is perturbed by less than 0.3. Here it is not the error in a particular number which is being measured, but rather the allowable error in all numbers for a computation to work correctly. Among other things, these predicates can be used to determine the amounts by which one would have to perturb data to account for rounding error in a perturbation scheme (see section 3). Unfortunately, there have been few predicates developed, and it is not clear how well this method will extend to more complex predicates.

Tolerances. Tolerances are one of the most common methods used in practice for increasing robustness. With tolerance methods, tests for equality are done within a certain specified tolerance. Thus, two points which are less than the tolerance value away from each other will be considered to be coincident. Or, two lines which are within the tolerance value of each other everywhere will be considered to be coincident.

There are several potential benefits to using tolerance methods ([Jac95]). First of all, by picking a tolerance value large enough, the error buildup due to floating pont arithmetic can be ignored for most cases. If the error buildup is less than half the tolerance, then comparisons between two values will still be valid. Another benefit to tolerances is that small features (those smaller than the tolerance) are automatically eliminated. In several applications, the presence of small features in the output can cause problems. A third benefit to the use of tolerances is that they can sometimes mimic actual physical constraints. For example, manufacturing tools are only able to handle a certain degree of precision - any higher precision in the output is wasted. Also, input data coming from CAD designs will often be created with tolerances in mind. It should be emphasized that the notion of tolerances for geometric computation (as a way to increase robustness) is different from, although related to, the technical notion of tolerances in CAD design. Design tolerances in CAD are tolerance amounts specifically placed by the designer in order to account for such things as wear or manufacturing imprecision ([FBZ93]). Work is also being done on ways to represent true design tolerances in geometric solid modeling systems ([GT93]) and in choosing appropriate design tolerances ([LJ93]).

At the same time, there are several drawbacks to the use of tolerances. First of all, unless a common tolerance value is used on all machines, there is no guarantee that the output from one machine will be the same as the output from another. Another drawback is that when a single global tolerance value is used for an entire computation, it is difficult or even impossible to find such a value. One part of the data might require a low tolerance and another require a high tolerance. No single tolerance will be appropriate for all requirements. Tolerances also suffer from the incidence transitivity problem which arises with interval arithmetic. Furthermore, error buildup can still occur to a point where the error is larger than the tolerance, defeating the whole purpose for setting a tolerance. Simply making the tolerance value larger will result in the elimination of small but possibly significant features, may make some operations, such as subdivision, more difficult, and may give output which is not as close as one would like.

One potential solution to some of these problems is the use of local adaptive tolerances by Segal ([Seg90]) and extended to non-planar cases by Jackson ([Jac95]). With local tolerances, different tolerance values are applied to different parts of the model. In areas where higher tolerance values are needed, they are increased as necessary, and topological structures which are close together are merged. This approach is similar to the use of interval arithmetic, in that the tolerances define an interval which is expanded as necessary to account for error buildup. The drawbacks to this approach are that several of the problems with tolerances remain, and in addition, the tolerance values can grow rather large.

An approach by Fang et al. also uses various tolerances for various parts of the model ([FBZ93]). In this approach, multiple tolerances are kept, and decisions are made as to whether geometric objects are apart, coincident, intersecting, or ambiguous (which requires refinement of the interval), based on how the tolerance bounds overlap. A potential drawback to this approach is that setting up and manipulating the tolerances may be very difficult. Still, this approach has been applied to boolean operations on 3D objects bounded by planes and quadrics, with promising results, so it may be useful for a number of other geometric calculations as well.

Data Normalization and Hidden Variables. Milenkovic uses the methods of data normalization and hidden variables to perform robust computation in finite precision arithmetic on linear objects ([Mil88]). With data normalization, the input data is modified to meet certain "normalization" conditions. Namely, points must be at least a certain distance apart and points must be at least a certain distance from a line. Points which are too close together are merged (this is "vertex shifting"), and edges which are too close to a point are split into two edges which meet at that point (this is "edge cracking"). The hidden variable method, on the other hand, does not actually modify numerical data, but operates as if the data were behaving in a certain way topologically. This behavior should be "close" to the real behavior but does not have to be exact. In effect, topological questions are answered as if the data were slightly different than it actually is, behaving according to some hidden topology. This hidden topology does not ever need to be computed specifically. Using these methods, more robust geometric algorithms are obtained. Milenkovic has extended his work with fixed precision data to begin looking at applications to higher degree objects ([Mil89]), but the application has been limited.

Rounding. Guibas and Marimont ([GM95]) have described a successful implementation of a method for rounding line segment arrangements to a grid. In other words, the endpoints and intersection points of the line segments are "snap rounded" to an integer grid. In many ways, this is similar to the data normalization approach used by Milenkovic. They show that this approach can be used to maintain the correct topology among all segments while representing segments with a relatively small number of segments with endpoints snapped to the integer grid. This technique holds promise as a way to consistently round geometric computations without altering the overall combinatorial structure.

Methods very similar to "snap rounding" have been used in to construct link paths. Link paths are a series of line segments connected at the endpoints. Minimum link paths for certain applications can be shown to require a greater number of bits of precision than the input data has. In order to determine a link path in finite precision, Kahan and Snoeyink determined a method for determining link paths where the endpoints were snapped to a predetermined grid ([KS96]). This allows a (not necessarily minimal) link path to be constructed without having to resort to higher precision (exact) arithmetic.

Estimated Error. Masotti ([Mas93]) has developed a method for performing floating point computation using an estimated error. When worst-case error estimates are kept (such as in interval arithmetic), one is guaranteed that the final result is located within a certain bounded interval. However, after several computations, this interval may be so large that it is worthless. Masotti's error estimate is much smaller than the worst-case error, and thus the final answer is more useful. In addition, this approach can detect certain ill-conditioned computations. Masotti demonstrates that this approach can be used to perform operations far more robustly than with standard floating point, although the results are still not guaranteed. Masotti's approach also has potential in that it can potentially be implemented in a hardware floating point processor.

2.2 Exact Computation

With exact computation, all computations are performed such that the outcome of the computation is known precisely. All numeric data is kept exactly, to whatever precision is necessary. This will mean that the size of the individual numbers must be taken into account when performing arithmetic operations on them. The real RAM model assumption that arithmetic operations take a constant amount of time is obviously not possible to achieve with exact computation.

There are numerous implementations of exact arithmetic routines. Exact arithmetic is used in most if not all of the computer algebra systems which are fairly common. Several programming libraries are also available for performing exact arithmetic. One of the most well-developed current libraries is the LEDA library ([ea95]). For a description of a number of other libraries available, see [YD95].

Exact arithmetic routines are usually confined to operations on integers or rational numbers. Some packages handle algebraic numbers as well, and this extension appears to handle almost all problems in computational geometry ([Yap94]). As long as computations require only rational numbers, exact arithmetic can be used directly in order to implement a geometric algorithm. Since the numbers and computations are exact, there is no need to worry about rounding error or the associated techniques used to compensate for it - tolerances, interval arithmetic, etc. The only real drawback to the use of exact arithmetic is its efficiency - numbers can take a significant amount of space to store and computations can take a long time to perform. Thus, most of the work in exact computation deals with ways of making the performance of exact arithmetic more efficient.

The "standard" method of representing exact integers is by storing a set of bits which exactly expresses the integer. Rational numbers are usually stored as two integers (for the numerator and denominator). Algebraic numbers are usually stored as two items: a polynomial equation and an interval. The interval is chosen such that there is only one real root of the polynomial within that interval. Thus the (possibly irrational) root is stored exactly, even though it is not represented as a series of bits. The interval size is selected so that no other roots of the polynomial lie in that interval. If desired, the gap theorem (see [Can88]) can be used to ensure that the interval is narrow enough that no other polynomials under consideration will have a root in the interval. Usually, however, that would produce an interval so small that alternate methods for ensuring isolated roots are explored. The idea of exact floating point numbers, where the mantissa and exponent are kept exactly has also been investigated. It should be noted that any representation which allows numbers to be stored exactly (or equivalently, such that they can be determined to *any* given precision) is acceptable in an exact computation paradigm.

Yap and Dube point out several reasons why exact computation is an important paradigm in the conclusion of [YD95]. Among the reasons listed there are that exact algorithms are much more straightforward than a corresponding "robust" floating point algorithm, that almost all algorithms (in computational geometry and elsewhere) are developed assuming exact arithmetic, and that exact computation is useful for studying and validating floating-point computation. Yap also points out in [Yap94] that as processor speeds increase, for certain problems, the appeal of robustness offered by exact computation may begin to approach the appeal of speed that floating-point offers. These papers ([YD95],[Yap94]) provide an excellent overview of many of the concepts and techniques used in exact arithmetic.

A number of techniques have been used in order to improve the efficiency or applicability of exact computation. Among these techniques are the following methods.

Interval Arithmetic. In [KLN91], Karasick et al. look into methods for improving exact computation by the use of interval arithmetic. In their approach, they attempt to perform computations using a simplified representation of the rational number (i.e. one requiring fewer bits). Basically, instead of computing with the rational number itself, they bound each rational number with an interval whose endpoints are rational numbers which require fewer bits. Interval arithmetic is used in the computation, and the final result will be an interval. For an appropriate predicate, such as sign of a determinant, the computed (interval) answer will either be enough to answer the question or will indicate that higher precision is needed. If higher precision is needed, then the original intervals are chosen to have more bits of precision. If all attempts at simpler intervals fail, the exact number itself can be used.

Since the interval bounds have far fewer bits than the actual number, the computation using interval arithmetic can potentially take far less time than the computation would have had the exact number (with all bits of precision) been used. The usefulness of this approach is limited, however, in that the predicates need to be carefully chosen so that they can be computed efficiently using interval arithmetic.

Another adaptation of interval arithmetic to exact computation is the approach taken by Johnson ([Joh92]). Johnson, like Karasick, uses intervals with rational endpoints. Using these intervals he shows how exact computations can be performed for real numbers which are bounded by an interval, even if the real number itself is not known. Examples of the computations which can be performed include sign evaluation and substitution into a polynomial. These computations can easily become significantly slower than "standard" rational number computations. In order to achieve some speedup, Johnson makes use of binary rational numbers, where the denominator is a power of two. Thus, only the power of the denominator needs to be stored, and basic rational operations ($+$, $\times$, etc). are faster.

Floating Point Filters. The idea of floating point filters was introduced by Fortune and van Wyk as a way to potentially avoid having to do any exact computation ([FvW93]). Like the interval arithmetic approach of Karasick et al., the idea is to compute an approximate value quickly, then determine whether that value is good enough. With floating point filters, the computation is performed in floating point, but an error bound is kept along the way. If the final error bound is too large to precisely determine the answer to some predicate, then the exact computation is performed. Otherwise, the answer determined by floating point is used. Fortune and van Wyk found that this filter could dramatically speed up the determination of certain predicates.

Tuning Computations. In the same paper ([FvW93]), Fortune and van Wyk propose a method for "tuning" a computation. By this, they mean that an entire computation for some predicate is parsed, and a complete straight-line program is created to evaluate that computation. The program is created to estimate the precision required for intermediate computations, and to then perform the necessary computation exactly. This program can then be compiled and run. Since the necessary bit complexity was already predetermined, the computation can be performed very efficiently. Fortune and van Wyk see significant improvements from this approach as well. They have recently implemented a package, LN (which stands for "little numbers"), which can automatically produce C++ code for evaluating geometric primitives ([FvW96]). This package includes floating point filters to increase efficiency.

Lazy Arithmetic. Another approach geared toward improving the efficiency of exact computation is the use of lazy arithmetic ([BMP94]). With this approach, a number is stored in a floating point interval format as well as in a directed acyclic graph (dag) format. The graph format is used to trace the "history" of how the final number is computed. For example, if two numbers are added together, the graph consists of two nodes, one for each input number, with links heading toward an addition node which represents the output. For most computations, only the interval is used. In case the intervals are not tight enough (e.g. the intervals of two numbers overlap when comparing them), the actual rational number answer can be computed by following the dag. Or, a progressive approach could be used, continually starting with tighter intervals and propagating those down (again resorting to the exact computation after a certain point). In the worst case, this will mean that just as many exact rational operations have to be performed. In the best case, exact rational operations never have to be performed, and the result can be determined from the floating point interval computations alone.

Precision-Driven Computation. Another approach, somewhat similar to the lazy arithmetic approach is to use Yap and Dube's precision driven arithmetic ([YD95]). With this approach, a dag is constructed, but instead of propagating information from the input to the output, the desired precision of the output is specified, and that information is propagated "backward" in the dag to determine what precision the input needs to be accurate to. In the worst case, the exact computation will need to be performed, but more typically, the input can be of less than exact precision and still give a final output which is within the desired precision.

Minimizing Intermediate Precision. Clarkson presents a method for finding the sign of a determinant exactly and efficiently with exact arithmetic ([Cla95]). He computes the result exactly, but manages to formulate the problem in such a way that the number of bits of precision required is far less than one would expect. This allows the approach to be implemented effectively using hardware-based arithmetic assuming that the original data contains a reasonable number of bits. By implementing a basic geometric predicate in such an efficient manner, an overall geometric operation may be performed in a reasonable amount of time, while maintaining exactness.

The drawback here is that the efficiency is obtained for only one operation - determinant sign evaluation. If similarly efficient approaches can be found for other important operations, then there is hope that exact arithmetic can be used widely and efficiently for many geometric problems. It is not clear, however, that such efficient implementations of other operations exist. Still, if a particular geometric problem can be formulated such that it requires use of only a single geometric predicate, then it may be worth the effort to find an implementation of that predicate which uses minimal precision for intermediate computation.

Fast Hardware Computation. Shewchuck has developed a method for quickly finding exact determinations of some predicates using only standard floating point hardware ([She96]). Shewchuck uses specialized routines for performing exact arithmetic using floating point hardware. An adaptive computation method is used such that predicates which can be determined with less than exact precision are. The impressive speed comes from the use of floating point hardware, the relaxation of storage conditions for high precision numbers, and the use of adaptive computation with the predicates. This approach is very promising in that exact computation can be performed more quickly, and if the desired predicates are amenable to it, adaptive computation can allow them to be computed quite quickly.

Modular Arithmetic One approach that is commonly used (e.g. [FvW93])to speed up exact computation is the use of modular arithmetic. Rather than storing integers as long series of bits, the integers are stored modulo a set of prime numbers with small, fixed precision.

Then, computation can take place in hardware for each of the modulo numbers. If necessary, the Chinese remainder theorem can be used to construct the final number. In some cases, the final number does not have to be completely reconstructed, since one might be interested only in some aspect of the number (e.g. the sign). In some cases, use of modular arithmetic has given significant speedup.

3 Degeneracies

Degeneracies arise whenever two given objects are not in general position. The types of possible degeneracies are numerous. For example, when dealing with point sets, possible degeneracies might include two points being coincident, three points being colinear, or four points being cocircular ([Yap90]). For some algorithms or problems, certain types of degeneracy might not matter. For instance, having four cocircular points might not matter in a problem dealing with convex hulls, while three colinear points would.

The use of naive fixed-precision arithmetic may cause degeneracies to occur where they should not. For example two points which are computed to be very close together may be considered to be coincident by a program. Then, the program will see this case as a degenerate one, even though the exact representation should not be.

Complicating matters is the fact that data which is in a degenerate configuration is often meant to be that way. For example, a CAD designer may position two objects so that their faces overlap (with the intention of having them welded together). The overlapping faces will cause problems inside of a routine which assumes that all faces are in general position. The use of floating point arithmetic can also cause such information to be lost - removing a degeneracy where there should be one. This would happen if, say, two parallel planes were rotated, and floating point rounding caused the two to be off by some small amount - just enough to make them not parallel.

The manifestation of degeneracy problems in an actual program is generally the failure of a predicate to determine which branch of a program to take. Predicates are generally the sign of some polynomial computation. When there are no degeneracies, the predicate determines which of two branches to take based upon whether the sign of the polynomial is positive or negative. When degeneracies are present, the predicate polynomial may evaluate to zero, thus making it unclear as to what branch of a predicate to take.

Two main approaches are used to handle degeneracies. These are the use of special cases and the use of symbolic perturbation.

3.1 Special Cases

Usually, computational geometry algorithms are designed assuming that no degeneracy exists. If a degenerate case does arise, it is expected to be treated as a ”special case” for the problem.

This would mean that a predicate is allowed to evaluate to zero, and some "special case" handling routine is called whenever one does. Although such an assumption can lead to a clear description of the algorithm, it can also make implementation far more difficult.

In order to implement such an algorithm, a programmer must determine all potential degeneracies, detect them, and then deal with them. Each of these requirements can be very difficult. It might not be clear what all the potential degeneracies are, detecting the degeneracies may be difficult and can lead to very inelegant (and confusing) code, and there might be no simple way to actually deal with a detected degeneracy within the framework of the original algorithm. Still, treating degeneracies as special cases is usually the most straightforward and most commonly used method of handling them. For one example of an attempt to handle degeneracies through the use of special cases, see [Yu91].

One final note should be made about the effect that inexact computation will have on this approach. With inexact computation, a predicate might evaluate to zero even when it shouldn't (effectively creating a degeneracy when there's not one), or a predicate which should evaluate to zero might not (effectively removing a degeneracy that should be there). Although in some cases the "loss" of a degeneracy might seem like a good thing, it may cause serious problems if the special case handling routines expect to find all degeneracies.

3.2 Perturbation Methods

One of the more promising approaches for dealing with degeneracies is the use of perturbation methods. The idea behind perturbation methods is fairly simple. Perturbation methods modify the input data in such a way that one is guaranteed to have no degeneracies. In effect, all that we are trying to do is to make sure that a predicate never evaluates to zero. There are a number of different techniques for doing so.

The use of perturbation methods at all is somewhat questionable ([Sei94]). One of the most compelling arguments against using perturbation methods is that degenerate cases, as mentioned earlier, are often there on purpose. Thus, perturbing the data to make it nondegenerate actually loses important information. Another strong argument against perturbation is that the solution which is determined is not actually the solution to the given problem. Thus, the computed solution might have to be transformed into the solution to the given problem. Such a transformation may be difficult. A third drawback to the use of perturbation methods is that the perturbed problem might be much more difficult to solve than the original problem. Also, the use of perturbation methods generally requires some form of exact computation in order to be successful. This will make the overall implementation run slower than one which did not use exact computation.

Despite these drawbacks, perturbation methods remain as one of the best known approaches for dealing with degeneracies as a whole. A well chosen perturbation method will eliminate all potential degeneracies, meaning that no special cases have to be detected. Since exact computation is being used, this means that algorithms can be implemented in their most straightforward sense. The use of exact arithmetic coupled with perturbation methods brings us closer to the real RAM model of computation than almost any other known approach. Perturbation methods have proven to be successful for several problems, and so there is hope that they can be applied in general to a much wider array of problems.

In a more formal sense, Seidel defines a perturbation scheme ([Sei94]):

For input space, $\mathcal{I}$ and output space, $\mathcal{O}$, with $q \in \mathcal{I}$, a *perturbation* of q is any curve $q() : [0, \infty) \mapsto \mathcal{I}$ with $q(0) = q$.

A *perturbation scheme*, Q, induces for every function $F : \mathcal{I} \mapsto \mathcal{O}$ a perturbed function $\overline{F}^Q : \mathcal{I} \mapsto \mathcal{O}$, defined by

$$\overline{F}^Q(q) = \lim_{\epsilon \to 0^+} F(q(\epsilon))$$

If F is continuous at q, then $F(q) = \overline{F}^Q(q)$. Even when F is not continuous at some q, we will often be able to easily recover $F(q)$ from $\overline{F}^Q(q)$. Thus, if our computation is continuous for all input, the output of the perturbed problem is exactly the same as that for the unperturbed problem. In cases where the computation is not continuous for all input (as may be the case for degenerate input conditions), we will need to recover the output for the original problem through some sort of transformation on the output from the perturbation scheme. In many cases, this transformation will be fairly simple, although it can conceivably be rather difficult, possibly even more difficult than solving the unperturbed, degenerate problem in the first place. It should be pointed out that the actual perturbation is never computed. Rather, the computation proceeds treating the perturbation amount as a variable, and the limit of the perturbation is taken at the end.

Following Seidel's outline, there are three primary proposals for perturbation methods.

Simulation of Simplicity. Probably the first of the general perturbation schemes proposed was the Simulation of Simplicity (SoS) scheme of Edelsbrunner and Mücke ([EM90]). With this method, given a set of n geometric objects, each of which takes d input parameters, the data is perturbed such that

$$\pi_{i,j}(\epsilon) = \pi_{i,j} + \epsilon^{2^{i \cdot \delta - j}}$$

for $0 \leq i \leq n-1$, $1 \leq j \leq d$, and $\delta \geq d$. Notice that the amount of the perturbation is different for every i, j pair. This ensures that each of the original points is perturbed to a different amount

This perturbation can then be used in some predicate. For example, the paper by Edelsbrunner and Mücke uses a point orientation test, which simply boils down to computation of the sign of a determinant. The determinant of the perturbed points gives a polynomial in terms of ϵ. Thus, to find the sign of the determinant as ϵ approaches 0^+, we need to find the sign of the lowest degree nonzero term of the polynomial. It is interesting to note that the polynomial's constant term (i.e. its lowest degree term) is nothing more than the determinant of the matrix without any perturbation applied. Thus, in effect, the only time that the perturbed case requires more work than the non-perturbed case is when the non-perturbed determinant evaluates to zero - i.e. a degenerate case. In these cases, the sign of the determinant is computed from the sign of the smallest nonzero term of the polynomial. It so happens that this term will be the determinant of some minor of the original unperturbed matrix. Thus, a sign determination can be made, even when the data is given in degenerate format.

Linear and Random Perturbation. Emiris and Canny present a variation to the SoS method which makes use of a linear perturbation ([ECS94], [EC94]). Specifically, the perturbation used is:

$$\pi_{i,j}(\epsilon) = \pi_{i,j} + \epsilon \cdot i^j$$

Notice that in this representation, the perturbation is linear in ϵ. This fact helps to improve the efficiency of the overall approach. Also, it means that the overall scheme can be viewed as a vector, π_i, being perturbed by another vector, b_i, where the elements of b_i are just i^j. Emiris and Canny show that this perturbation scheme can be used in a variety of predicate tests, and even apply it to a convex hull computation. They perform a rather extensive analysis of time and bit complexity required.

Another approach presented by Emiris and Canny in the same papers is a random permutation of the form:

$$\pi_i(\epsilon) = \pi_i + \epsilon \cdot r_i$$

In this representation, r_i is a randomly chosen number from a large interval. This perturbation is more efficiently evaluated (assuming the random numbers are chosen with an appropriately smaller number of bits!) than the earlier one, but it is not guaranteed to eliminate all degeneracies. Instead, the probability of still having a degeneracy is extremely low. In the unlikely event that a degeneracy remains, it can always be detected and the program restarted with a new random choice for a perturbation amount. This method is particularly applicable to cases where the computation is for branching on some arbitrary rational function, as opposed to branching on a determinant, which the earlier method is more suitable for.

Symbolic Perturbation Yap's method for symbolic perturbation is far more general than the other perturbation methods ([Yap90]. In Yap's formulation, a predicate can be *any* polynomial or rational function, or, according to [Sei94], even an analytic function. Yap has a predicate return a number if that number is not zero, or, if the predicate value is zero, return either 0^+ or 0^-. This is really no different than simply returning the sign of the predicate, just as the other methods do. With Yap's method, a large polynomial is produced, and from it a series of polynomials is found by taking a number of derivatives. I will not go into the details of this process here, but the final sequence of polynomials is used to determine the returned value.

If the polynomial itself evaluates to zero, then either 0^+ or 0^- is returned based upon the sign of the first nonzero polynomial in the sequence. Although this approach is very general and powerful, it may be difficult to compute the potentially high derivatives and large polynomial.

3.3 Other Approaches

Besides perturbation methods and handling degeneracies by treating them as special cases, a few other approaches to dealing with degeneracies exist. In most cases, these methods are used

Redundancy Elimination. Rossignac and Voelcker have used a method called redundancy elimination which can eliminate sources of degeneracies ([RV89]). With redundancy elimination, data is preprocessed to remove information which is not needed in order to solve problems based on that data (i.e. redundant information). Although the primary focus of such work is for faster implementations, removing some of this extraneous data can eliminate some degenerate cases (e.g. when an object is unioned into another object it is already a part of in a CSG operation).

Automatic Parsing An approach taken by Farouki et al. toward dealing with certain types of degeneracy is automatic parsing ([FNO93]). With this approach, an expression is set up to determine when a degenerate case can occur, and the case is automatically formulated as a non-degenerate one instead. In this way, several "cases" can be eliminated by a relatively simple transformation. For this approach to work, the degenerate case must involve some expression which can be turned into an equivalent expression which yields a non-degenerate case. In Farouki's example, this involves factoring a quadratic polynomial. This can be a very useful approach in cases where it applies, but usually there is no simple way to create an equivalent non-degenerate case.

4 Conclusion

Robustness problems continue to be one of the major obstacles faced in implementing geometric algorithms. These problems stem from the fact that algorithms are designed with the unrealistic assumptions of computation using the real RAM model and all input data given in general position. The failure of these two assumptions gives us our two broad categories for robustness problems - problems with precision and problems with degeneracies.

Problems with precision are dealt with by either finding ways to make floating point operations more robust or by using exact computation. Although exact computation is conceptually simpler, it can lead to a tremendous loss in efficiency. Because of this, a large amount of effort has gone into finding ways to make exact computation reasonably efficient. Some of the most promising approaches are those which attempt to minimize the actual amount of exact computation performed by using cheap floating point computations when possible. A large amount of work has also gone into trying to make floating point operations robust. A number of approaches have been developed, with tolerance methods being some of the most likely to be used in practice, probably due to their ease of use.

Problems with degeneracies are usually handled by treating degeneracies as special cases. Numerous problems with this approach have led to the development of perturbation methods, which eliminate degeneracies and thus come closer to meeting the assumptions that many algorithms are based upon. Although perturbation methods have their downside, they remain the most promising methods for dealing with degeneracies in general.

The "robustness issue" is still a relatively new area of computational geometry interest, having arisen only in the last decade or so. It is safe to assume that as more and more cases occur where robustness problems arise, interest in this subject will continue. There are still be plenty of opportunities available for those wishing to improve the robustness of floating point computation, make exact computation more efficient, find efficient methods of perturbation application, or investigate new methods for handling degeneracies.

References

[BMP94] M. O. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: A detailed implementation. *Computer Aided Design*, 26(6):403–416, June 1994.

[Can88] John F. Canny. *The Complexity of Robot Motion Planning.* The MIT Press, Cambridge, Massachusetts, 1988.

[Cla95] Kenneth L. Clarkson. Save and effective determinant evaluation. Manuscript, February 1995.

[CS93] Joao Luiz Dibl Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. Anais do VII Sibgraphi, 1993.

[dF96] Luiz Henrique de Figueiredo. Surface intersection using affine arithmetic. In *Proceedings of Graphics Interface '96*, pages 168–175, 1996.

[DSB92] Tamal K. Dey, Kokichi Sugihara, and Chandrajit L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.

[ea95] C. Burnikel et al. Exact computation in leda. In *Proceedings of the 11th ACM Conference on Computational Geometry*, pages C18–C19, 1995.

[EC94] Ioannis Z. Emiris and John F. Canny. A general approach to removing degeneracies. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, pages 405–413, 1994.

[ECS94] Ioannis Z. Emiris, John F. Canny, and Raimund Seidel. Efficient perturbations for handling geometric degeneracies. Manuscript, June 1994.

[EM90] Herbert Edelsbrunner and Ernst Peter Mucke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[FBZ93] Shiaofen Fang, Beat Bruderlin, and Xiaohong Zhu. Robustness in solid modeling: A tolerance-based intuitionistic approach. *Computer Aided Design*, 25(9):567–576, September 1993.

[FNO93] R. T. Farouki, C. A. Neff, and M. A. O'Connor. Automatic parsing of degenerate quadric-surface intersections. *ACM Transactions on Graphics*, 8(3):174–208, July 1993.

[For95] Steven Fortune. Voronoi diagrams and delaunay triangulations. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 225–265. World Scientific Press, Singapore, 1995.

[For96] Steven Fortune. Robustness issues in geometric algorithms. In *Proceedings 1st ACM Workshop on Applied Computational Geometry*, pages 20–23, 1996.

[FvW93] Steven Fortune and Christopher J. van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th ACM Conference on Computational Geometry*, pages 163–171, 1993.

[FvW96] Steven Fortune and Christopher J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, July 1996.

[GM95] Leonidas J. Guibas and David H. Marimont. Rounding arrangements dynamically. In *Proceedings of the 11th ACM Conference on Computational Geometry*, pages 190–199, 1995.

[GSS89] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the 5th ACM Conference on Computational Geometry*, pages 208–217, 1989.

[GT93] James Guilford and Joshua Turner. Representational primitives for geometric tolerancing. *Computer Aided Design*, 25(9):577–586, September 1993.

[HK89] John E. Hopcroft and Peter J. Kahn. A paradigm for robust geometric algorithms. Report TR 89-1044, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[Hof89a] Christoph M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[Hof89b] Christoph M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computar*, 22(3):31–41, March 1989.

[HPY96a] Chun-Yi Hu, Nicholas M. Patrikalakis, and Xiuzi Ye. Robust interval solid modelling part i: Representations. *Computer Aided Design*, 28(10):807–817, October 1996.

[HPY96b] Chun-Yi Hu, Nicholas M. Patrikalakis, and Xiuzi Ye. Robust interval solid modelling part ii: Boundary evaluation. *Computer Aided Design*, 28(10):819–830, October 1996.

[Jac95] David J. Jackson. Boundary representation modelling with local tolerancing. In *Proceedings of Solid Modeling '95*, pages 247–253, 1995.

[Joh92] J. R. Johnson. Real algebraic number computation using interval arithmetic. In *Proceedings of ISSAC '92*, pages 195–205, 1992.

[KLN91] Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[KS96] Simon Kahan and Jack Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. In *Proceedings of 12th Annual ACM Symposium on Computational Geometry*, pages 151–158, 1996.

[LJ93] J. Lee and G. E. Johnson. Optimal tolerance allotment using a genetic algorithm and truncated monte carlo simulation. *Computer Aided Design*, 25(9):601–611, September 1993.

[Mas93] Glauco Massotti. Floating-point numbers with error estimates. *Computer Aided Design*, 25(9):524–538, September 1993.

[Mil88] Victor Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. Report CMU-CS-88-168, Department of Computer Science, Cornell University, Pittsburgh, July 1988.

[Mil89] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the 5th ACM Conference on Computational Geometry*, pages 197–207, 1989.

[RV89] Jarolsaw R. Rossignac and Herbert B. Voelcker. Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Transactions on Graphics*, 8(1):51–87, January 1989.

[Sch96] Stefan Schirra. Precision and robustness. *Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, Italy*, September 1996.

[Seg90] Mark Segal. Using tolerances to guarantee valid polyhedral modeling results. In *Proceedings of SIGGRAPH 90*, pages 105–114, 1990.

[Sei94] Raimund Seidel. The nature and meaning of perturbations in geometric computing. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *STACS 94*, volume 775 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, Berlin, 1994.

[She96] Jonathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the 12th ACM Conference on Computational Geometry*, pages 141–150, 1996.

[Yap90] Chee-Keng Yap. Symbolic treatment of geometric degeneracies. *Journal of symbolic Computation*, 10:349–370, 1990.

[Yap94] Chee-Keng Yap. Towards exact geometric computation. Manuscript; To appear, Computational Geometry: Theory and Applications, November 1994.

[YD95] Chee Yap and Thomas Dube. The exact computation paradigm. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 1995.

[Yu91] Jiaxun Yu. Exact arithmetic solid modeling. PnD. Thesis, Purdue University, December 1991.