GEOMETRY, GRAPHICS, & NUMERICAL ANALYSIS

Deborah E. Silver

(Thesis)

CS-TR-178-88

October 1988

GEOMETRY, GRAPHICS, & NUMERICAL ANALYSIS

Deborah E. Silver

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

OCTOBER 1988

Acknowledgements

I wish to thank first and foremost my advisor David Dobkin. He has been a constant source of ideas, advice, encouragement, guidance, and most of all patience during the course of my stay at Princeton. He introduced me to the subject matter, and has always made himself available. For all of this, I am enormously grateful and appreciative.

I am indebted to my readers, Andrew Appel and Bernard Chazelle, whose comments and suggestions undoubtably improved the quality of this dissertation, and to Ken Supowit for the helpful advice he has given me. Further thanks goes to Sharon Rodgers, Gene Davidson, Gerree Pecht, Grace Kehoe, Ruth James, and Winnie Waring for making the administrative details painless, and for always being extremely helpful and cheerful. I am obliged to the csstaff for enabling my research to continue with relatively few interruptions.

I would like to thank my officemates (especially Susan), my fellow graduate students, the faculty members, and all my friends at Princeton who have made my stay here both fruitful and enjoyable. A special thank you goes to my sister, Naomi, for wading through a preliminary version of this thesis. Needless to say, I am grateful to the ''Department of Computer Science'' and Princeton University for granting me the opportunity to study computer science in such a wonderful atmosphere. It is a gift I will cherish long after I leave.

Some of the work presented here has appeared in the paper ''Recipes for Geometry and Numerical Analysis; Part I: An Empirical Study'' [Dobk88a].

This research was supported in part by NSF grant CCR-87-00917 and by a Hewlett-Packard Faculty Development grant.

to my parents

Abstract

Geometric computations (e.g. polygon/line intersections and detection, etc.), like all numerical procedures, are extremely prone to roundoff error. However, virtually none of the numerical analysis literature directly applies to geometric calculations. As a result, most of these applications resort to *ad hoc* methods to overcome the numerical difficulties. The goal of this research is to present and analyze systematic approaches to handle unreliability and enable more robust solutions to problems in applied computational geometry.

Table of Contents

Part I

"Close only applies in horseshoes, hand grenades, and numerical analysis." - popular saying

1

Introduction

Introduction

1. General Introduction & Motivation

The earliest computers gave the user the ability to increase the speed of arithmetic computation a hundredfold over conventional methods. With the speed increase came the problems of limited precision and roundoff error. As computation has matured, these problems have grown, and a science has developed to study them. This science, numerical analysis, seeks to differentiate ''theory'' from ''practice''. However, there have been few applications of numerical analysis to geometric computations. Our goal here is to extend this science to that realm.

Along with the advent of high speed computation to solve complex mathematical problems, there came the need to analyze the errors inherent in the finite nature of such computations: namely, the study of how to handle the limitations imposed by the existing computer hardware in terms of time and space. After mathematical theories and functions were transformed into efficient algorithms, the difficulty of applying those algorithms became apparent. While the theory assumes perfect calculations, the practical world of computers can only deal with finite precision calculations. This causes the numerical inaccuracies and inconsistencies inherent in a finite precision domain to hamper the employment of many otherwise elegant algorithms.

The quest for higher precision is never satisfied. As computers become more powerful and provide larger floating point capabilities, scientist, programmers, and users demand even more.

Floating point calculations are never accurate enough (and may never be). There is always a tradeoff between accuracy, efficiency, and memory. While larger precisions become widespread, applications requiring greater resources are developed. Algorithms using sophisticated mathematical techniques are stymied because the inaccuracies of computer calculations become overwhelming. As more theoretical algorithms become applied, new difficulties arise stemming from the finiteness of the computational facilities. The new applications are always pushing the boundary of what is available.

This gap between the real world and the theoretical one impedes further progress. As the algorithmic techniques mature, attention must be focussed on the problem of **technology transfer**. The goal is to determine which theoretically fast algorithms work well in practice and to find methods of turning the efficient into the practical. The switch is to ''low level'' issues to find methods utilizing the available hardware in the best way possible and hopefully limit the devastating effect of error accumulation.

1.1. Computational Geometry

While some of the older fields of computer science have learnt to manage numerical ''inconveniences'' (operations research, numerical methods ...), many of the newer areas are just beginning to recognize it as a major setback. Geometric procedures, i.e. procedures pertaining to geometric computations, comprise one such area. The geometric primitive and more complex processes are those operations performed on objects, and include such fundamentals as locus testing, object intersection detection, and intersection calculation. This group forms the basic primitive operations of computational geometry, computer graphics, and geometric modeling. However, few of these fields have dealt with the actual numerical calculations and errors stemming from them. Unfortunately, much of the work in numerical analysis is not directly applicable to geometric problems, as well as being difficult to understand and implement. The specifications for many geometric procedures are not the same as those for standard numerical processes. The output is different, topological constraints must be met, and the sequence of operation is varied. Numerical analytical techniques are just one part of this complex puzzle.

As more geometric applications become widespread, the ''holes'' in these system become visible (figuratively and literally) and the ''ad hoc'' methods formerly used to control the disastrous effect of roundoff error lose their effectiveness. With the increase in computational power now available, the feasibility and demand for ''correct'' and better results exists. The need for a systematic and methodical approach to *fill the gaps* and handle the *exceptional* cases is evident. Verification of results for many computations should not just be an admirable feature but an essential element of many of these systems.

However, the theory is still lacking. Most models of computation assume that arithmetic is done flawlessly. This is precisely expressed by the following quotation:

> *''As is the rule in computational geometry problems with discrete output, we assume all the computations are performed with exact (infinite-precision) arithmetic. Without this assumption it is virtually impossible to prove the correctness of any geometric algorithms.''* [Mair87a]

Unfortunately, that assumption is seldom valid in the real world. Roundoff error plagues all computation intensive procedures, and geometric algorithms are no exception. Thus, the central problem of the technology transfer lies in the generation of fast algorithms which are robust. The definition of robust, according to Webster's dictionary, is "full of health and strength; vigorous; hardy", and that is exactly what should be expected from any numerical algorithm. Basically, the computed output should be *verifiably correct* for all cases. ''Correct'' here is a relative term subject to the application. For graphical output, anywhere between three to twelve significant digits of precision will suffice (depending upon the display), whereas for other areas, more may be needed. Having a program compute twenty digits of precision when only three are needed is overly costly and time consuming. Verification of output is equally important. How many significant digits are in the result or, how much error has accumulated in the computed values? If a good error estimate can be calculated easily, then a program can essentially correct itself and tailor its precision. Needless to say, the program should handle all cases and, if it is unable to compute an answer, should inform the user instead of generating a random answer or dumping core.

The need for more user control is obvious. Allowing the user to specify the desired end precision, instead of the program arbitrarily selecting one, should be an option. In a similar vein, giving the user more control in manipulating the "work load" of the program will enable efficient use of available resources. A robustness measure, maybe between zero and ten, can be specified for each execution of the system, indicating the effort that should be expended in working towards a solution. The low end of the scale signifies a ''do-nothing'' execution of the program, whereas the high end of the scale could mean an accurate execution of the program at the expense of efficiency. A middle value would be somewhere between the two extremes; the result would be reasonably accurate and generated within a reasonable time period.

Another important issue is that of cost. How much extra does it cost to implement a robust system? Does the complexity of the algorithm change, or is it just a constant multiple applied? Assuming, of course, that it is possible to write a 100% accurate and reliable system.

However, this is not a trivial issue. Forrest [Forr85a, Forr87a] argues that there are no robust algorithms for even the simple and basic problem of line segment intersection . The recent flurry of activity on the problem confirms his belief. Knott and Joe [Knot87a] present a method for robustly determining if two line segments intersect and for computing their intersection, and there are cases where robustness costs as much as a factor of 100 in speed! Compounding the difficulties, many graphics and geometric algorithms perform iterations of calculations reusing computed results as input for subsequent calculations. Not only must each individual computation be robust, but the whole series of calculations must be robust as well. Ultimately, the errors become apparent by producing visible *glitches* in picture outputs or causing program failures when the computed topology becomes inconsistent with the underlying geometry (see Figure 1).

In addition to proposing solutions to some pressing problems of unreliability, it is the goal of this research to explore what is involved in generating robust geometric computations. In the remainder of this chapter, these issues will be explained further.

2. What are geometric computations?

Geometric computations encompass a wide range of numerical calculations. Namely, those techniques and procedures relevant to the computer manipulation of geometric objects. The objects include vertices, edges, faces, curves, patches, and solids, in any combination. These calculations occur in many different fields of computer applications, such as:

1. Computational Geometry: Although this field also deals with the processing of geometric objects, much of the classical work done in computational geometry is theoretical. (This type of research, dealing with the practical issues of computational geometry in the real sense, can be termed *applied* computational geometry.)
2. Computer Graphics: the manipulation, creation, and rendering of objects.
3. Computer Aided Design / Computer Aided Manufacturing: the creation and manipulation of geometric objects, including such topics as solid modeling, etc.
4. VLSI: circuit layout, design rules checking, routing.
5. Robotics: object avoidance, transformations, grasping.
6. Pattern recognition, vision: convex hull techniques.
7. Statistics: data analysis, data visualization.
8 Scientific visualization.

All of the fields listed above use geometric primitives and procedures in one way or another [Guib87a].

2.1. Geometric Primitives

The basic geometric primitives are computations that are performed on and with objects and aid in the creation and rendering of the objects. The functions include:

1 Locus testing: Calculating the general *location* of an object; three basic procedures are involved,

   i. Location determination: Computing the location of an object either alone or with respect to other objects. For example, testing whether a point is to the right, left, or on a particular line. *[1]* This type of test can be performed between points, lines, planes, surfaces, curves, etc., as in the case of *incircle testing* (determining whether a point is inside a circle formed by three other points).

   ii. Distance and angle determination: Computing the distance or angles between objects. Also the area of objects, unit normals, perpendiculars, etc.

   iii. Point determination: Calculating the points on a curve or surface (to represent them on some display, etc.)

2. Intersection calculations: Calculating the intersection region of two objects: such as, line segment intersections, line object intersection, solids intersection, curve intersection, surface intersections, etc.

3. Transformations: Transformations on objects done for viewing and for correct object orientation: scaling, rotation, translation, perspective.

[1] see appendix

These are just some of the primitive functions that form the backbone of many more complex geometric processes. For example, clipping, object merging, and hidden surface elimination would all require locus testing and intersection determination. See [Mudu86a] for more detail on geometric computations.

3. Robustness

The need for *robust* geometric systems is apparent. But what exactly does *robust* entail? Forrest states [Forr87a],

> *"... computational geometry raises systems engineering issues which need both theoretical and practical attention. We must be able to develop efficient, robust, reliable, accurate, consistent systems which are documentable and maintainable."*

However, the question of what exactly that means is still unclear. In order to implement robust geometric systems it is important that the qualities of such a system and the needs of the users be understood.

3.1. User's Needs

A programmer or user has certain basic requirements for a system. These are fundamental features that enable any program or system to be useful. Namely, the program should

1. handle all cases
2. produce correct (reliable) output
3. not fail

Needless to say, it should also be efficient.

Although some of the properties appear similar or redundant, any one does not include the other two, and a system without all three is still unreliable. Furthermore, a fault in one of the areas may have repercussions in another, so the requirements are not necessarily distinct.

As any programmer knows, implementing a system containing these characteristics is not trivial. For each application, different types of errors arise that contribute to the unreliability of a program. In order to produce robust systems, such obstacles must be overcome. It is also important to note that these are ''ideal'' attributes - they occur in the ''ideal case''. Real world programs can strive for perfection, but rarely attain it. Fortunately, most users are satisfied as long as the three requirements are handled in a somewhat reliable manner. For example, if a program fails, it should fail gracefully, as opposed to dumping core without a warning or an error message. Similarly, numerical errors could be flagged and the users notified if such errors may cause trouble later on.

3.2. Geometric Robustness Issues

In geometric computations, the problems that arise are directly analogous to the basic requirements which they cause to be violated. The three basic issues (in opposite order to the three listed above) are

a. Topological Consistency

b. Numerical Accuracy

c. Comprehensive Case Analysis

The issues are all interelated, and all cause different types of errors to surface during implementation and execution of geometric procedures.

3.2.1. Topological Consistency

Topological consistency refers to the maintenance of geometric objects in a *consistent* state while they are being operated upon. Because many of the primitive operations performed on objects change the original objects and create new ones, inconsistencies can arise (mainly caused by numerical inaccuracy, see below). For example, if calculated intersections do not lie on the originating object, co-planarity of the vertices of a polygon in 3-space is lost, adjacent objects become separated, non-intersecting objects overlap, etc. While floating point computations are largely to blame, simply increasing the numerical accuracy may not necessarily remove the problem (see the example of the incidence test in the next chapter). Topological inconsistency can lead to erroneous results and program failure.

3.2.2. Numerical Accuracy

Numerical accuracy is lost when finite precision is used. Because the majority of computer applications use floating point arithmetic, roundoff error results and the precision of intermediate calculations suffer. This loss affects the topological consistency of the objects, as well as the output (picture *glitches* in graphic applications). It also contributes to the difficulty in handling *degenerate cases* (see below).

3.2.3. Comprehensive case analysis

Many theoretical algorithms only deal with the *general* cases and do not provide for (or consider) *degenerate* cases. These instances occur if the input to a function is in some way degenerate. For example, during a sort operation, a degenerate case arises when the two quantities to be sorted are equal. (One way to handle the situation is to keep equal elements in the same relative order in the sorted sequence as they were in the original sequence [Horv74a].) For geometric procedures, theoretical algorithms may assume that no points are collinear, lines are not vertical, etc.. Furthermore, *topological primitives* test a given input and classify it as one of a number of possible cases, one or more of which may be *degenerate.* However, the definition of a *degeneracy* depends on the problem being considered. (See the next chapter for an example.)

A further difficulty for degenerate cases is detecting them. In many geometric primitives a value of zero after a sequence of calculations signifies a degeneracy has occurred. However, with the use of floating point calculations an exact value of zero is highly unlikely to be computed. Instead a *small number close to zero* may result. The bound on the small number is dependent upon the input numbers, the set of calculations performed, and the precision used.

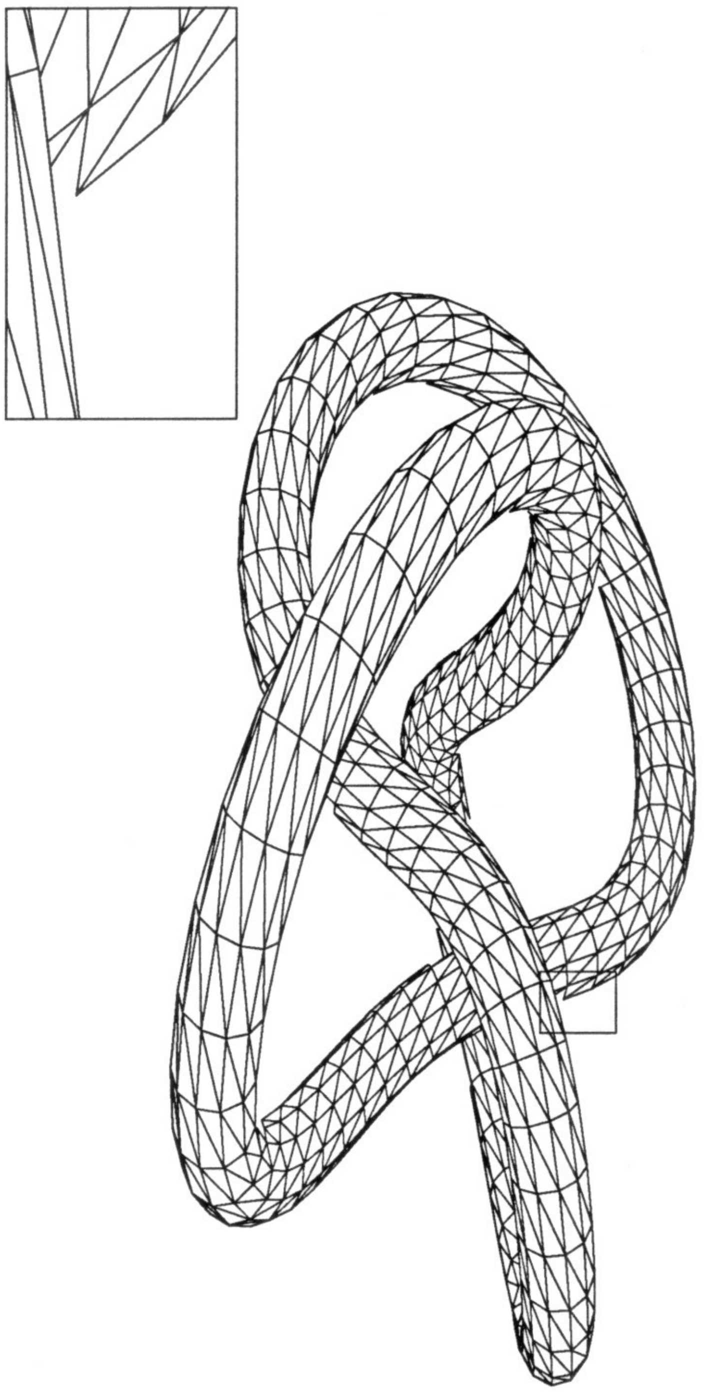


Figure 1. Picture Glitch

4. Robust Geometric Systems

It is evident that a geometric system must account for all three error factors in order to provide the basic requirements and be labeled ''robust''. These characteristics should apply to the geometric primitives, as well as, to the higher level procedures. A system is as robust as its weakest component.

Most implementers are painfully aware of the lack of robustness for almost all the geometric computations. Much of the work in the various fields of computer science that use geometric functions has been concentrated on developing algorithms, paying little, if any, attention to the details of implementing them. As stated before, Forrest argues that there are no robust (and efficient) algorithms for the basic line segment intersection function [Forr85a, Forr87a].

It would be beneficial for the programmer (or theoretician) to have a basic set of robust tools to use when writing geometric procedures, thereby permitting larger systems to be built from reliable building blocks. A programmer should be able to simply consult a ''handbook of robust geometric computations'' for any geometric problem which arises in the course of an implementation (see [Pres86a]).

In this thesis, the aim is to explore some of the issues in robustness and geometric computations, to provide a systematic method to calculate and handle errors for certain types of problems, and lay the groundwork for additional research. While it is beyond the scope of the thesis to provide a complete handbook (if that is possible), it is the ultimate goal of any research in the area, and hopefully this work will become a part of that effort.

5. Thesis Outline

The thesis is divided into three parts. Part I (chapters one and two) is a basic introduction to the area of robustness and reliability in geometric computations. As it is a new field of interest, a comprehensive introduction and motivation was presented in this chapter. Chapter two contains a survey of the current literature and proposed solutions to the problems of unreliability. Part II presents a specific type of geometric computation, and discusses some of the robustness issues that arise. A solution to the specific problem is proposed, and the problem and solution are implemented. Extensions of the solution, as well as its applicability to other problems, are also presented. The last chapter contains a conclusion and some ideas for future work. Part III contains the appendices.

2

Literature Survey

Literature Survey

1. Error Type

Unlike other types of numerical computations, in geometric and graphic computations roundoff error is visible (literally). In other areas, errors become apparent when program failure occurs. This is also true in geometric computations, but as an added ''benefit'' the roundoff error is eye-catching as well. It is, in fact, difficult to ignore the effect of roundoff error in many graphical computations. What is surprising though, is that the methods of dealing with this problem are varied and no formal robust approach exists.

Roundoff error in geometric and graphic computations manifests in two basic ways:

1. The underlying topology becomes inconsistent and causes program failure (or if the program contains tests for these events, the tests return positive).
2. The errors produce ''glitches'' in picture outputs, on a screen or any other display device.

In the literature, attempts have been directed towards studying one and/or both of these problem areas. However, correcting one without the other can lead to erroneous conclusions. While the picture may look correct, the underlying topology may not be as accurate, because the output resolution is much lower than the object (internal) resolution. What you see may not necessarily be what you have [Forr87a]. The reverse is also true. If care is not exercised when rounding to screen resolution, the gaps and ''glitches'' in the pictures become noticeable (as happens when viewing successive frames) or can display information which was not intended and is wrong (see the example of *spurious intersections* later in this chapter).

2. Growth of Error

Before attempting to overcome roundoff error in a systematic and robust way, it is necessary to specify the cause of error growth during geometric processes. The four fundamental reasons are the following:

a. The input data (object coordinates) may contain roundoff error. The objects are sometimes approximated by simpler surfaces; for example, curves may be approximated by lines (surfaces by polygons) etc. Or the data may have been computed by another program which introduced error during its numerical computations. (The input data is therefore just an approximation to the actual information.)

b. Graphics programs are generally calculation intensive (number crunchers) and the computations produce errors. In general, the more computations performed, the more likely the error resulting from these calculations will present difficulties. (See Chapter 1 for more detail.) In addition, much of the geometric data contains degeneracies which cause further numerical difficulties.

c. Rounding to the screen resolution or grid device can cause errors, as stated before.

d. The drawing algorithms used by the graphic or display device may introduce error (for example, line drawing or circle drawing algorithms). See [Bres87a] for more detail.

A further characteristic of the geometric world is that it is primarily based in two domains, namely, *raster* or *image* space (external resolution for the output device) and *geometry* or *object space* (internal resolution). The raster space is typically of lower resolution. Most of the proposed ''solutions'' or ideas to limit roundoff error attempt to correct one or more of the problem areas concentrating in a particular domain, and some of the solutions project from one space to the other to resolve the numerical difficulties.

Roundoff error is handled by either estimating it or attempting to limit its growth. The magnitude and effect of roundoff error can be controlled with certain actions, such as increasing precision, and many of the solutions use this tactic. However, continually increasing precision, for example, can be an ''unending'' process as well as costly in terms of time and space. The second type of solutions involve estimating the amount of error or possible error that can occur in the computations. Calculations can be done to estimate the amount of error in a set of computations or to detect if the computations are sensitive to roundoff error (using condition numbers). When error estimation is performed, action must be taken (such as notifying the user or increasing precision) if error is predicted or if the magnitude of the error is ''too great''. (Solutions to mode one may be used.)

In addition to these two approaches to roundoff error, it is helpful to quantify the solutions to the problem of robustness in geometric procedures in terms of two other classifications. Although geometric computations are really just numerical computations, some of the proposed solutions attempt to exploit the special topological features of the objects for error control. This class of solutions is labeled *geometric flavored solutions* and have dealt mainly with maintaining correct topological information using finite precision. The *numerical flavored solutions* adapt numerical analytical techniques for the geometric procedures and functions in order to increase the precision and accuracy of the computations.

3. **** The Most Popular Solution is ....

However, the most widely applied solution to the problem of roundoff error in geometric computations (as well as other computations) is the *ad hoc* approach: calling the local guru to pull a fix out of his/her magic box. This usually entails arbitrarily increasing precision, reordering calculations, tweaking specific numbers, or arbitrarily selecting *epsilon* and *fuzz* values (the small floating point numbers put into programs to compensate for numerical inaccuracies), and in most instances, will only solve a set of problems temporarily, because it does not attack the underlying cause of the roundoff error. Needless to say, this approach is far from robust and consistent.

4. Geometric Flavored Solutions

4.1. Raster Mode

Since geometric computations are done often for viewing purpose, many of the proposed solutions perform the calculations in grid space (grid arithmetic) thereby avoiding floating point computations. (The typical approach is to perform floating point computations and then round to screen space.) Taken to an extreme, objects are stored as integer screen coordinates and set operations are performed on the ''objects'' or ''sets of pixels''. For example, in Corthout and Jonkers [Cort87a] an algorithm is presented to determine *point containment* (whether a point is contained in a mathematically defined region) for B_Regions in the discrete plane (closed paths of connected Bezier curves). Unfortunately, only a few limited applications can use this approach.

Greene and Yao [Gree86a] attempt to bridge the gap between the discrete and continuous domain. In their work, objects are transformed from the continuous domain to the discrete domain and all the calculations are performed in the discrete domain. In this way, they hope to avoid the invalid solutions that may arise when the properties of the finite-resolutions space are not taken into consideration. For example, running an ordinary intersection algorithm and then rounding to the nearest grid point may produce undesirable results; *spurious intersections* can arise if an intersection is rounded and that rounded value happens to coincide with another line creating the illusion of an intersection which, in fact, was not originally present and should not be there.

For the problem of line segment intersection, lines are confined to a set of raster points (these are the points used by line drawing algorithms) and the line is the shortest path within the *envelope* of points. The line-path is controlled with hooks which serve to direct the line to pass through specific grid points insuring that it will intersect certain lines while not crossing others. The hooks are introduced when line segment intersections are found (either a line segment with another line segment or a line segment with a hook on a line segment). For instance, if line segments l_i and l_j intersect at p, then the hook (p,q) is added to both lines (the representations of those lines), where q is the closest grid point to p. Hence, when the lines are drawn, both will pass through the point q and will appear to intersect, as they should. The shortest line-path through the envelope of points is calculated using continued fractions.

4.2. Object Space

In this category, procedures that attempt to utilize the features of geometric objects are applied to the objects in order to avoid numerical difficulties. Many of the techniques have special functions and data structures to keep the geometric objects in a ''consistent'' state. *Epsilon procedures* are used to handle ambiguous cases and keep the objects ''far enough'' apart. Objects can be thought of as having a magnetic field surrounding them; if objects are ''close'', then they are ''attracted'' to each other and merged together. In contrast to the above methods, the objects are not specifically discretized to the raster space, although the epsilon values may, in fact, be associated with the output specifications.

4.2.1. Tolerance Region

To overcome the numerical inaccuracies on topological decisions in solid modeling, Segal and Sequin [Sega88a, Sega85a] maintain tolerance regions with all vertices, edges, and faces to detect ambiguous cases. Tolerances are assigned to each vertex and face reflecting the accuracy of the coordinates that describe the object. The *tolerance region* of a vertex is a spherical ''fuzz'' area centered about the vertex. The region for an edge is a truncated cone based on the tolerance regions of its two endpoints. For a face, the region is a half width of an infinite slab centered about the faces plane. The tolerances are called *minimum feature size* and *face thickness*. A *consolidation* procedure merges or pulls apart objects which lie within each others tolerances, thereby insuring a minimum distance separation between distinct primitives. Transformations applied to objects must be checked so that the minimum feature size requirement is not violated. If it is, then consolidation is performed.

During this procedure, if ambiguous or illegal situations result, a *tolerance error* is signaled. For example, if two vertices lie within a tolerance of one another then they are merged into a single vertex with a larger tolerance. If the greater tolerance exceeds the *maximum allowable tolerance* then a tolerance error has occurred. Similarly, if faces are deemed coplanar, then they are merged. Ambiguous situations can also emerge when topological datum is computed. When determining the position of a vertex relative to a face plane, the vertex lies in the face plane if and only if its tolerance region is entirely within that of the face. If the regions partly overlap, then an arbitrary decision is made as to whether to include the vertex or not, and a tolerance error is signaled. The tolerances lessen the effect of some of the numerical obstacles which can arise. However, the use of tolerances cannot resolve all the possible ambiguities, and in such cases, the user must provide more sophisticated procedures and data structures to insure consistent output. The method detects such situations.

4.2.2. Data Normalization

In a similar vein, Milenkovic [Mile86a] proposes two methods for verifiable implementations of geometric algorithms using finite precision: *data normalization* and the *hidden variable method.* In *data normalization* the structures and parameters of the objects are altered slightly in order to keep the objects in a ''consistent'' state in which all the numerical tests are provably accurate. The objects and operations on these objects must satisfy a set of normalization rules, thereby insuring a valid system. The normalization rules include maintaining a distance at least ε, determined by machine roundoff error and the number of arithmetic operations per expression between the geometric structures (between vertices and between edges and vertices which are not connected). Because the basic operations performed on polygonal regions can violate the normalization rules, *accommodation* operations are performed which force conformation by altering the objects. These operations are called *vertex shifting* and *edge cracking*. Vertex shifting merges two vertices which are closer than ε apart. Edge cracking subdivides edges which have vertices within a distance of ε away; the edge is ''cracked'' into two edges with the offending vertex as an endpoint to both new edges). Milenkovic proves that the accommodation procedures terminate, and he presents worst case bounds for the maximal positional perturbation. Unfortunately, normalization may displace an edge many times and introduce large deviations from the ''real'' result.

4.2.3. Hidden Variable Method

The second method is called the *hidden variable method*, which constructs configurations of objects belonging to an infinite precision domain without actually representing those infinite precision objects. The topology of the infinite precision objects is known but not their numerical values. The method can be used to determine the topological arrangement of a set of lines represented by their line equations (how they intersect in a consistent manner within a bounding box). This is done by modeling approximations to geometric lines with *xy-monotonic curves.* A curve α is monotonic with respect to a non-zero direction vector v if the inner product of v and $\alpha(t)$ is either non-decreasing or non-increasing with t. Intuitively, the curve tends in the v direction and does not backtrack. Curves are *approximately monotonic* if they do not backtrack more than ε.

The hidden variable method has a lower level system, which models xy-monotonic curves, and a higher level system which models approximately monotonic curves and uses the lower level system as a subroutine. In the lower level system, the vertex locations and topological structure are derived so that the curves can be modeled. Line intersection is performed first using finite precision arithmetic. If an intersection within the ''known rectangle'' cannot be guaranteed, then an incremental algorithm is used to find the intersection point within the rectangle. (If the lines L_1 and L_2 have positive slope and two points v_1 and v_2 are known, such that v_1 lies above L_1 and below L_2, and v_2 lies below L_1 and above L_2, then the intersection point of the two lines must lie within the rectangle defined by v_1 and v_2.)

In the higher level design, hidden curves are approximately monotonic. In effect, the higher system solves a transformed problem in which each line is replaced by a pair of lines a distance of ε apart (the ε here is larger than the ε of the lower level system). The resulting arrangement can be thought of as a set of *tiled highways* (each line becomes a strip or ''highway'' and the intersections of the strips form ''tiles''). The paths along the highways represent the approximate monotonic curves, and these paths cross at vertices in the arrangement. This approach can be extended to a polygonal region modeling system as well. The error introduces in such a system would be dependent on the number of lines and not the number of operations.

4.2.4. Symbolic Reasoning

Hoffmann, Hopcroft, and Karasick [Hoff88a, Hoff88a, Kara88a] add symbolic reasoning to compensate for numerical uncertainties when performing operation on geometric objects, forcing any new decision to be consistent with previous ones. The data in most geometric applications consists of two components: topological information describing incidence relations, and numerical information containing a numerical component (face equations defined with floating point numbers). The notions of *representation* and *model* are used to comprehend the relationship between the two components. A *representation* is a description of a geometric object with fixed precision, possibly using imprecise numerical data. A *model* is the ''infinite'' precision object that satisfies the symbolic part of the description exactly (the question of whether there exists a model for a given representation is not trivial). Because the numerical data is subject to roundoff error, basing a decision on it alone may lead to inconsistent results. Therefore, the numerical information is treated as an approximation to the real data, and the symbolic data describing the topological constraints is used to derive information and conclusions. For example, in Figure 1. the incidence of line-segment *e* with the intersection of planes *P* and *Q* ($P \cap Q$) is computed. If *v* is in *Q* but not in *P*, and *u* is not in *Q* (does not lie on the extended plane), then the edge *e* cannot intersect $P \cap Q$ (the opposite case is also true) even though, with finite precision, *a* may appear to intersect $P \cap Q$. (If only one of *u* or *v* lies on Q, then *a* cannot intersect $P \cap Q$. This can be seen more clearly if *u* is slightly above the extended plane *Q*, and *v* on it. Then the edge *e* cannot intersect *P* except at *v*. Similarly for *u* slightly below *P*, although in this case, the edge *e* will intersect *Q*. Only when both *u* and *v* are on *P* will the line between them intersect $P \cap Q$.)

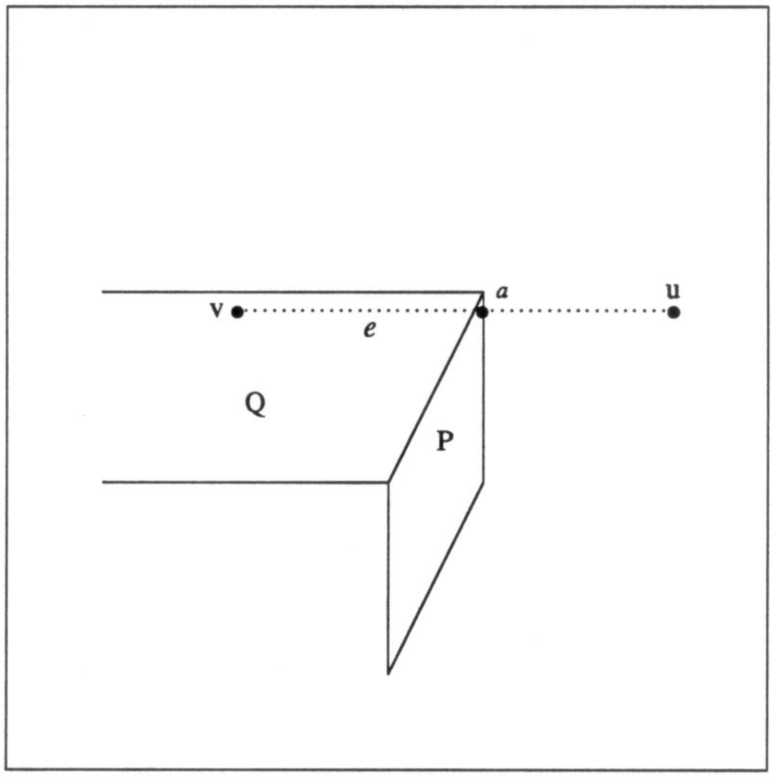


Figure 1. Incidence test

In their work, the concept of *minimum feature separation* of a boundary representation is used to guarantee robustness. Vertices are separated by a certain tolerance, edges are larger than a minimum length, and the angles between faces or edges are not greater than or less than certain specified values.

To determine if object features are incident, a numerical computation is done. If the features are separated by more than ε, then they are not incident. However, deciding that the objects do coincide cannot be concluded from the numerical computations alone. Tests must be performed to insure that any decision is not inconsistent with a previous decision or fact, i.e. the implications of the features being incident do not contradict other known properties of the objects. If no such contradiction occurs then the features are ''deemed incident''.

Interestingly, the reasoning paradigm varies with the input. Intersecting two polygons requires little reasoning; however, for the simultaneous intersection of three or more polygons, theorems from projective geometry must be considered.

4.2.5. Symbolic Computations

Going to a greater extreme, all the calculations can be performed using symbolic representations avoiding numerical problems completely. The ''symbols'' or objects (e.g. points or lines) are declared in advance, and the incidence relations are expressed as logical statements which must be satisfied. However, some logical existence problems may occur if the statements are contradictory. (See [Hoff88b] for more detail.)

(On a side note, symbolic computation is useful for geometric editing. Ericson and Yap [Eric88a] describe a constraint based editor for correct topological editing, without errors due to numerical approximations. See also [Buch88a].)

4.2.6. Braiding Lines

On a slightly different tone, Ramshaw [Rams82a] discusses the phenomena of *braiding lines*. When floating point numbers are used to represent line segments, the errors which arise because of floating point arithmetic can make the lines appear to intersect more than once. If the lines are thought of as functions which return a y value for every input x, two lines may braid when there is sequence of three values $x_0 < x_1 < x_2$, such that, at both x_0 and x_2, the first line is strictly higher than the second, while at x_1 the second is higher than the first. Ramshaw presents many examples of this anomaly, suggesting that any straightforward implementations of the line function using floating point arithmetic may cause braiding. A possible solution to the problem is presented, which uses the Slope-Intercept Formula with the final multiply and add evaluated in double precision. The final result is then rounded back to single precision.

5. Numerical Flavors

The most obvious numerical solutions approach the use of infinite precision. This is attempted by using rational arithmetic, integer arithmetic, or multiple precision arithmetic (not limited to the hardware capabilities of a particular machine). These functions are generally implemented in software. Once chopping is added to control the size of the numerical information, such as limiting a denominator or numerator, roundoff error will result. However, increasing precision will, in many instances, bypass numerical troubles in specific situations.

Sugihara [Sugi87a] has proposed to use exact rational arithmetic of bounded precision to implement operations on polyhedral solids. In his representation, only plane equations are given numerically and all other information is symbolic. The operations on primitive polyhedra are trihedral and satisfy a minimum feature condition. Precision is limited by putting bounds on the coefficients of the plane equations. However, translating or rotating a complex polyhedron can cause difficulty if the polyhedron contains many small features. Problems also arise in generalizing to nonlinear elements since the intersection points may not be rational. (See [Hoff88b])

There is also some work in implementing exact real arithmetic (in functional languages) using the *lazy evaluation* paradigm for optimization [Boeh86a, Schwa]. Real numbers are formulated as potentially infinite sequences of digits evaluated on demand, and optimization is achieved by avoiding the recalculation of digits that were previously computed.

In general, the numerical flavored solutions apply classical numerical analysis to geometric computations. Most of the geometric procedures are just numerical calculations and can therefore benefit from the well-studied field of numerical analysis, specifically, roundoff-error analysis. Pioneered by Wilkinson [Wilk63a], roundoff error analysis involves forward and backward error analysis and computing condition numbers for sets of calculations. Condition numbers measure the sensitivity of the solution to small perturbation in the input and can alert the programmer or user to possible bad sets of data and unstable algorithms [Mill80a]. A by-product of the condition numbers are some common-sense issues, such as reordering calculations to avoid ''undesirable'' calculations (adding together very large and small numbers, subtractive cancellation, etc.). (See the appendix on floating point arithmetic for more detail.)

5.1. Scalp Function

Slightly modifying the standard mode floating point system is another popular approach. Kulisch and Miranker [Kuli83a, Kuli86a, Kuli81a] introduce a dot product function that computes the dot product of two vectors rounding only at the end instead of after each individual multiplication and addition (the function is called *scalp(x,y)* for vectors *x,y* of size *n*). The result is therefore the correct computation of a scalar product, and is known as an *lsba-evaluation* of a function (*l*east *s*ignificant *b*it *a*ccurate) since it is the best approximation to the "real" value of the function using floating point computations. It is represented as ($\Box$ denotes rounding to the nearest machine number):

$$scalp\,(x,y) = \Box \sum_{i=0}^{n} x_i * y_i$$

Ottmann, Thiemt, and Ullrich [Ottm87a] show how to implement "stable" geometric primitives with this dot product function. For example, in determining intersecting pairs of line segments, the intersection test can be reduced to a location test which can be evaluated with the *scalp* function. Intersection determination can be performed similarly. However, it is necessary to have access to the original data as well as the computed results to maintain ongoing correct lsba evaluations.

5.2. Interval Arithmetic

A very popular numerical technique to estimate the error of floating point computations is interval arithmetic [Moor66a]. An interval is an ordered pair of real numbers:

$$[a,b] \quad a \le b$$

An unrepresentable floating point number, x, can be represented by its bounding representable number, enabling the amount of possible error to be calculated (for $x \in [a,b]$, $b-a = 1ulp$ [1]). Further computations are performed on the intervals, widening the resulting interval as necessary. (An exact real number can be represented by a degenerate interval $[a,a]$.) The interval

[1] *u*nit in *l*ast *p*lace, see appendix

computation encompasses the exact real number computations.

Interval arithmetic is governed by the following rules:

$$[a,b] + [c,d] = [a+c, b+d]$$
$$[a,b] - [c,d] = [a-d, b-c]$$
$$[a,b] \times [c,d] = [\mathit{min}(ac,ad,bc,bd), \mathit{max}(ac,ad,bc,bd)]$$
$$\frac{[a,b]}{[c,d]} = [a,b] \times [\frac{1}{d}, \frac{1}{c}] \text{ when } 0 \notin [c,d]$$

(Both addition and multiplication are commutative and associative, but the distributive law does not hold.)

Implementing straightforward interval arithmetic for mathematical functions generally results in loose bounds or large intervals. Madur and Koparkar [Madu84a] apply interval arithmetic to the processing of geometric objects and attempt to narrow the computed interval of some common geometric procedures. Several geometric functions are defined with interval computations, and new algorithms are devised using these functions for such tasks as curve drawing, surface shading, and intersection detection. Parametric ranges (which define curves and surfaces in graphics programs) are manipulated with interval methods. Recursive subdivision techniques are used to implement the common geometric tasks (intersection, renderings), and special functions are given to tighten the bounds on the interval computations.

Knott and Jou [Knot87a] also use interval arithmetic to *determine correctly whether two line segments intersect*. In their approach, a two stage process is used. In the first stage interval arithmetic is used to calculate the intersection (or to verify the presence of one). If the interval arithmetic fails to verify an intersection ''cleanly'', then the calculations are performed using a multiple precision package with enough precision to insure a correct result. ''Cleanly'' refers to the accuracy of the parametric values s and t. [2] If s and t are strictly contained in either the open interval $(0,1)$ or the open intervals $(-\infty,0),(1,\infty)$ then no further processing is needed (in the first case the segments intersect and in the second they do not). However, a problem arises if s,t contains either 0 or 1 in its interval. In this case, the intersection of the two segments cannot be determined with just interval arithmetic and higher precision must be used. To accurately compute the intersection, a precision of $2(h+k+f)$ bits are needed, where f is the number of bits in the mantissa, and k and h are the exponent ranges. They estimate that the interval computation is about 5-10 times more costly than direct single precision computation, while the multiple precision computation is 100-200 times more costly than direct computation (mainly because the procedures are implemented in software). In [Knot87a] code is given for both the interval arithmetic and the multiple precision computations.

5.3. Permutation Perturbation

A slightly different approach is that of Vignes and La Porte [Vign74a]. Although similar to interval computations, this method evaluates the number of significant digits in a computed result by stochastic means. A subset of all the possible computable results of a function is generated (in the interval) and the subset is used to determine properties of the entire set. The method is referred to as the *permutation perturbation method* or as *CESTAC* (controle et estimation stochastique d'arrondi de calcul), and its detail is described below.

Because of perturbation and permutation, the results of a set of mathematical computations performed on a computer are not unique. *Perturbation* refers to the rounding of a computed

[2] Given two line segments defined by the points: $\overline{P_1P_2}$ and $\overline{P_aP_b}$ the parametric equations are $f(s)=(1-s)P_1+sP_2$ and $f(t)=(1-t)P_a+tP_b$. The two lines intersect if $0 \le s,t \le 1$. See the appendix for more detail.

value up or down when being assigned to a variable. Therefore, any arithmetic computer operation can have one of two valid answers (one by lack, the other by excess); if an algorithm has k operations, there are a possible 2^k values. Since computer operations are not associative, rearranging the arithmetic operations in an algorithm may generate different results; this is known as *permutation.* (See the appendix on floating point arithmetic for an example of non-associativity of computer operations.) Let C_{op} be the total number of different possible permutations of the operators in a particular algorithm. When permutation and perturbation are applied in all possible combinations, the total set, R, of different computable solutions to a particular function, can be derived. R is of size $2^k \times C_{op}$.

Let R_0 be the computed result of an algorithm, where r is the real result. The absolute error of R_0 is just

$$\varepsilon_r = R_0 - r.$$

If $\tilde{\varepsilon}_r$ is the mean estimate of ε_r, $\bar{R}$ the mean value of all the elements of R, and δ the standard deviation of these elements, then

$$\begin{aligned}\varepsilon_r^2 &= (R_0 - r + \bar{R} - \bar{R})^2 \\ &= (R_0 - \bar{R})^2 + 2(R_0 - \bar{R})(\bar{R} - r) + (\bar{R} - r)^2\end{aligned}$$

Since r (or the best floating point approximation to r) can be any element in the population R, then [3]

$$\tilde{\varepsilon}_r^2 = (R_0 - \bar{R})^2 + \delta^2.$$

[3] since $2(R_0 - \bar{R})(\bar{R} - \frac{\sum_{r \in R} r}{|R|}) = 0$ and $\frac{\sum_{r \in R} (\bar{R} - r)^2}{|R|} = \delta^2$.

Therefore,

$$\tilde{\varepsilon}_r=\sqrt{(R_0-\bar{R})^2+\delta^2}$$

and so

$$\frac{\tilde{\varepsilon}_r}{|R_0|}=10^{-C}$$

is the relative error of R_0 where C is the amount of significant decimal digits in the result R_0.

It is virtually impossible to generate all the elements of the set, thus only a subset (3 or 4 elements) of the total population is used to compute the number of significant digits and this has been shown to be sufficient [Mail79a]. (For more complete detail, examples, and theory see [Vign78a, Faye85a, Vign74a].)

5.3.1. Example

The following example illustrates the CESTAC method: Given the four numbers to be added using four digit floating point arithmetic:

$$.1025\times10^4 \quad -.9112\times10^3 \quad -.9773\times10^2 \quad -.9315\times10^1$$

By reordering the sequence of addition the following three values can be computed:

$$.6755\times10^1$$
$$.6685\times10^1$$
$$.68\times10^1$$

(The real answer is $.6755\times10^1$.) The average of the three computed values is $.6747\times10^1$. The standard deviation is $.58\times10^{-1}$. Dividing the average by the standard deviation and taking its logarithm yields 2.06. The amount represents the number of significant digits in the average of the three computed values. (Notice that $.6747 \times 10^1$ has two significant digits.)

Part II of the thesis will demonstrate the application of this method to geometric procedures.

6. Degeneracies

The third of the basic contributing factors to the difficulty of robustness in geometric computations (listed in Chapter 1) is handling *degenerate cases*. An implementor of topological primitives must account for these special cases, and it can considerably complicate the programming effort. The canonical example illustrating case degeneracy is the *parity algorithm* used to solve the *point-in-polygon test*. The problem is to detect whether a point, v, lies inside a closed polygon P (both the point and polygon are given). The parity algorithm constructs a horizontal half line, with left endpoint v, and counts the number of intersections the half line forms with the polygon P. If the number is odd, then v is inside P, otherwise, if the number of intersections is even then v is outside the polygon P. However, the algorithm does not account for many of the degenerate cases that can arise (disregarding points lying on the polygon boundary). And it is because of these special cases, that the implementation of this seemingly simple algorithm is not trivial [Forr85a]. The degenerate cases which occur are illustrated in Figure 2. For the parity algorithm to work, cases (a) and (c) must be counted as intersections, whereas cases (b) and (d) are not [Edel88a].

Both Yap [Yap88a] and Edelsbrunner and Mucke [Edel88a] propose ways to cope with the problem of degeneracies and to avoid having to iterate all the cases. Basically, the objects are perturbed slightly to remove the degeneracy. Care must be taken to make sure that the perturbations are consistent and do not introduce further degeneracies into the remaining objects. For example, referring above to the *parity algorithm* of the *point in polygon test*, perturbing the point v in the

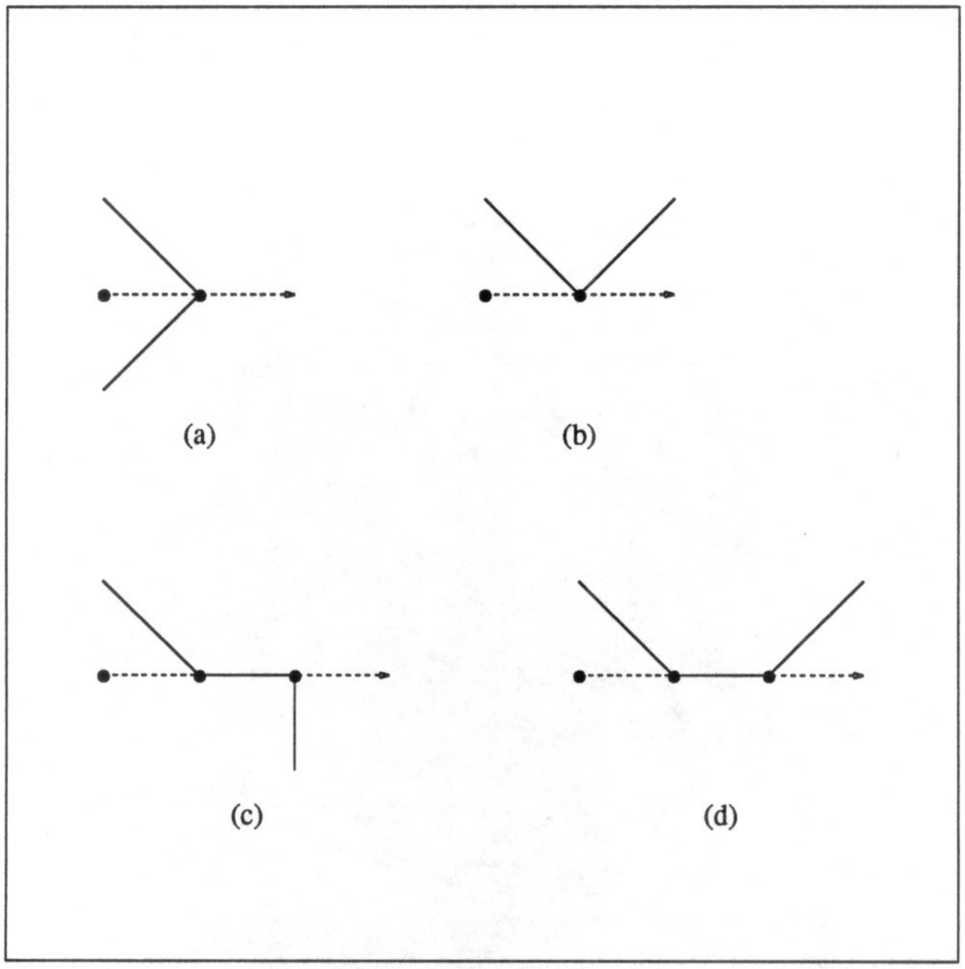


Figure 2. Parity Algorithm: Degenerate Cases

four degenerate cases of Figure 2 will give a correct intersection count. For cases (a) and (c), perturbing v results in one intersection, i.e. the point is *inside* P. For (b) and (d), perturbing the point will result in either zero or two intersections, an even number, signaling that v is *outside* the polygon P.

In these methods, the perturbations are never actually performed but done symbolically by replacing each coordinate with a polynomial in ε. The polynomials are chosen so that the perturbed set approaches the original set as ε goes to zero. The perturbation can be thought of as a black box that performs the topological primitives and always returns a non-degenerate answer, thereby removing the need to program special cases and significantly simplifying the programming that must be done. (See [Edel88a] and [Yap88a] for the theory behind the perturbation scheme and the programming details.)

7. Review

The solutions presented here all fit into the robustness framework of Chapter 1. (Recalling from before, the three main issues are topological consistency, numerical accuracy, and comprehensive case analysis.) Greene and Yao's method lies partially in the first two areas since it limits precision (by rounding to screen space) to maintain visually correct line intersections. Segal-Sequin, Milenkovic, and Hoffmann-Hopcroft-Karasick are primarily concerned with the topological consistency of the underlying object data base. The numerical procedures (roundoff error analysis, Knott-Jou, Ramshaw, Ottmann-Theimt-Ullrich, etc.) concentrate on the accuracy and precision of the data so that the results can be computed properly.

Because the purpose and function of geometric systems vary, sometimes one of these areas will be more important for the application at hand. However, all three must be accounted for to insure complete robustness and reliability.

Part II

"I could have done it in a much more complicated way," said the red Queen immensely proud. - Lewis Carroll

Sir, In your otherwise beautiful poem, The Vision of Sin, there is a verse which reads

"Every moment dies a man
every moment one is born."

Obviously, this cannot be true, and I suggest that in the next edition you have it read

"Every moment dies a man
every moment $1\frac{1}{16}$ is born."

Even this value is slightly in error but should be sufficiently accurate for poetry.

- Charles Babbage in a letter to Lord Tennyson

Part II

In the previous sections, the general area of robustness and reliability in geometric and graphic procedures was presented, consisting of an introduction to this relatively new field, motivation for further study, and a literature survey of previous work in the area. In this section, a specific type of geometric computation will be introduced. The implementations of a sample geometric problem displaying the characteristics of this type of computation is analyzed, and a robust solution is proposed. Chapter four contains an extension of this method and application to other geometric procedures. The last chapter presents some open problems and a conclusion.

3

Cascading Calculations

Cascading Calculations

1. Introduction

Many graphics and geometric algorithms perform iterations of calculations reusing computed results as input for subsequent calculations. Not only must each individual computation be robust, but the whole series of calculations must be robust as well. *Cascaded calculations* refers to the reuse of calculated data. This is similar to a series of waterfalls where the water passes through the first waterfall and cascades into the second, etc. Many computation intensive procedures have such cascading characteristics. A distinguishing feature is that intermediate values may be used and so cannot be bypassed for computing ''future'' calculations (e.g. to display partial results graphically).

While cascaded calculations suffer from standard roundoff error, they also suffer from *propagation error*. These are errors which arise from evaluating functions with inexact or approximate data. The propagation error, E_p, is

$$E_p = |F(X) - F(\hat{X})|$$

where $\hat{X}$ is an approximation of the actual data X. [1] Calculating with ''bad'' data will cause errors even when the calculations can be performed accurately. Therefore, if the computations are

[1] see the appendix on floating point arithmetic

not exact, cascaded calculations suffer from both propagation and standard floating point roundoff error.

For instance, the calculated point of intersection of two line segments may be used as a vertex of another line segment. Since this vertex is ''rounded'' and not ''exact'', the next series of calculations involving this point cannot be ''exact'', not necessarily because of the roundoff error generated from this particular set of calculations, but because the data is wrong. The ''real'' endpoint may be above, below, or to the side of the calculated one. Thus the calculated line is a shifted version of the ''real'' one causing any further computations with that line segment to be faulty no matter how exact the arithmetic functions are (of course, if all the calculations are precise this problem does not exist). This scenario is demonstrated in Figure 1. Line segment L has the vertex a as one of its endpoints. Because a is a calculated intersection point, it can be off because of roundoff error, causing L to be shifted, as shown in (1.B). A further problem is detecting the relationship between a and any other object. Referring to the figure, the question, ''is a above or below line m?'' cannot be answered accurately because it is very sensitive to the amount of error in a.

As calculations progress in an iteration series, the line segments are continually shifted and the final results may be nowhere near the true results. Ultimately, these errors become apparent by producing visible *glitches* in picture outputs or causing program failures when the computed topology becomes inconsistent with the underlying geometry. [Rams82a, Sega85a, Mile86a]

In what follows, some examples of cascading sequences of calculations in common geometric procedures are presented.

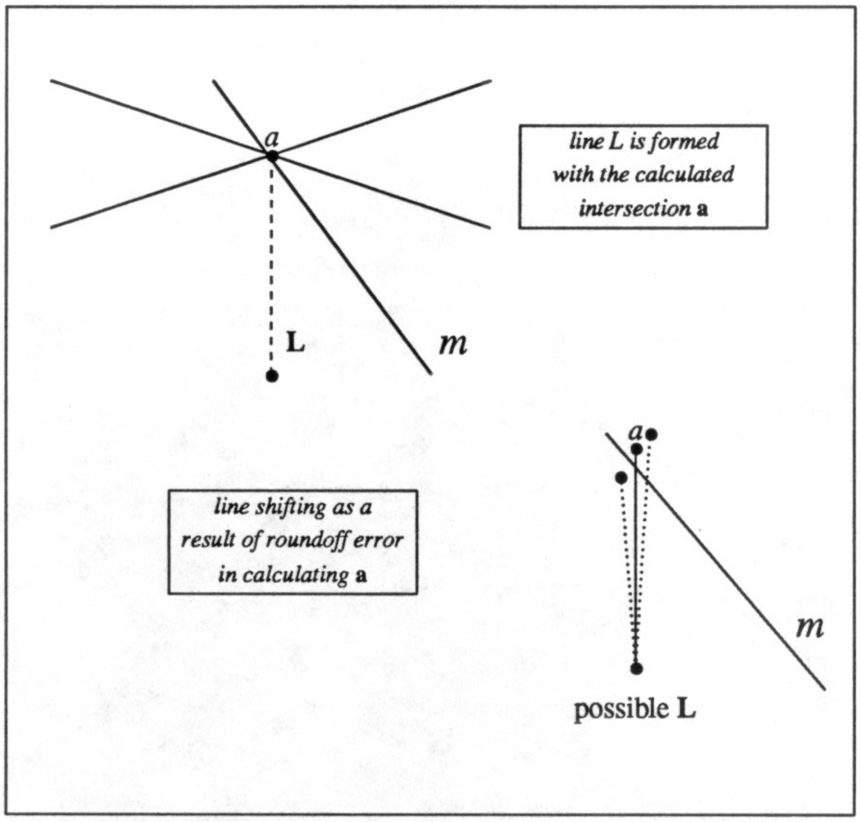

a
line L is formed with the calculated intersection a
L
m
a
line shifting as a result of roundoff error in calculating a
m
possible L

Figure 1. Line shifting

1.1. Examples of cascaded procedures

In several object space hidden surface elimination algorithms, [2] polygon intersection is

[2] e.g. Weiler-Atherton Algorithm [Roge85a]

performed by calculating the intersection of the *computed intersection of various polygons* with other polygons. These polygons may again be subdivided resulting in even ''smaller'' polygons. This process is repeated until all of the polygons are distinct, or can be displayed properly. In each subsequent pass, new objects are created from old ones as well as from new ones, using freshly computed values. This is demonstrated in Figure 2 where the original triangle is broken up by intersecting it with other objects. If one value is wrong, then all the polygons formed with that data will also contain error.

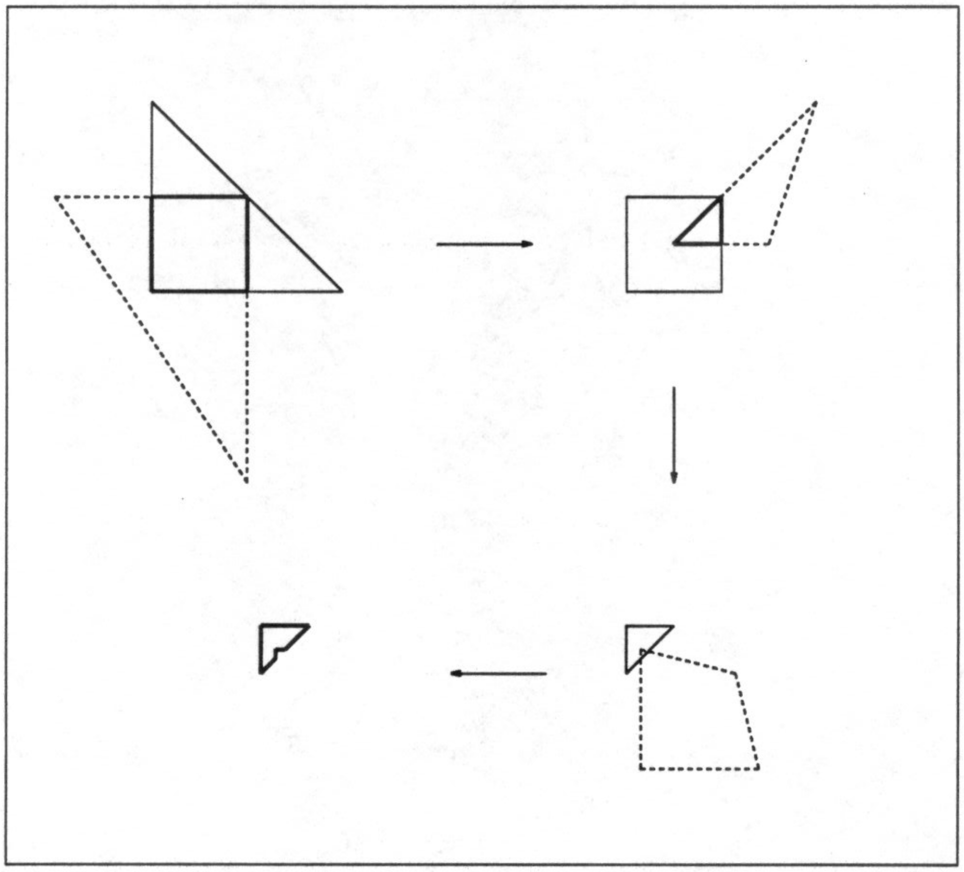


Figure 2. Cascading Polygon Intersection

1.1.1. Ray Tracing

Cascading calculations also occur in ray tracing. A ray is shot from the view point through the image plane and its intersection with an object is found (if one exists). If that object is reflecting or refracting, another ray is projected from this calculated intersection point, determined by the laws of reflection or refraction. The intersection of the new ray with other objects is computed, and this cascading process is continued if the objects hit by these rays are reflecting etc. (see Figure 3).

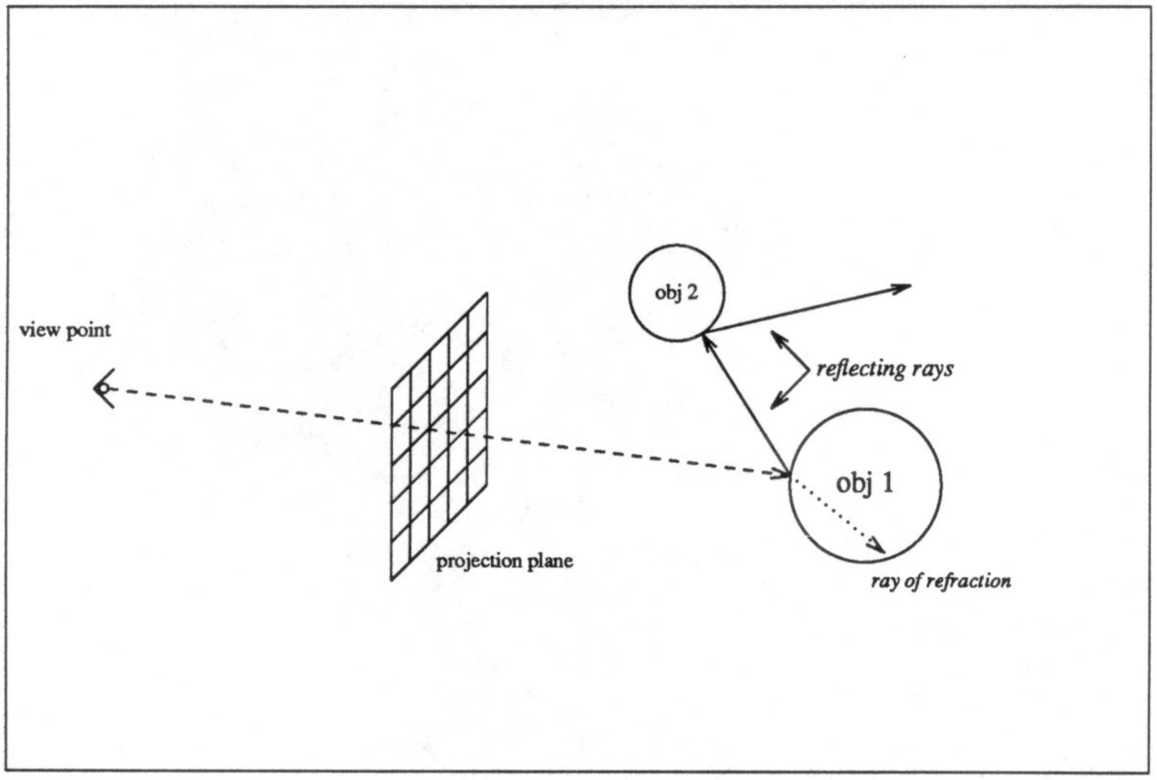


Figure 3. Ray Tracing

1.1.2. CAD/CAM

Computer aided design and manufacturing systems also contain cascaded calculations. As seen in [Trus88a] in discussing the intergration of CAD/CAM, one key issue is that of the accuracy of geometric information, in particular,

''As a first step towards integration, there must be complete confidence in the accuracy of any geometric data input and re-used by other activities. However, a problem that can occur with the accuracy of such information is that after a series of 'edits' or transformations (for example moving, scaling, rotation etc.) on geometric data, computational rounding errors will accumulate and can eventually alter a geometric size so much that, when displayed or visualised, it may exceed the specified manufacturing tolerance limits. The effect of such rounding errors on downstream activities is important.''

2. Solutions

What can be done to handle the roundoff error that results from cascading calculations? As with any type of roundoff error, there are two methods one can use: attempt to avoid the error completely, or compute the amount of error and then take action. However, both of these methods pose difficulties when applied to cascading calculations because of the extent of the cascade (which may not be known in advance) and the complexity of the entire set of calculations. Although useful in various situations, many of the methods discussed in the literature survey (Chapter 2) are of necessity flawed when applied to cascaded calculations.

While the geometric flavored solutions may be helpful for some types of cascaded calculations (topological based, as in solid modeling) they are not applicable to the calculations requiring numerical accuracy of the output (e.g. ray tracing, since reflection/refraction properties must be correctly modeled, and the *pentagon problem* which is discussed later in this chapter). A further difficulty is that these solutions may displace the objects from their original positions which is not desirable when an exact position is needed.

The numerical solutions are equally fraught with difficulties. Rational arithmetic and symbolic computations are slow, and the numerators and denominators (represented by numbers or symbols) grow very large very fast. Condition numbers do not give an accurate description of the exact accumulation of errors as the iterations increase, in addition to being difficult to calculate if all the cascading iterations are taken into consideration (as opposed to computing the condition number of a particular set of calculations, while ignoring the unknown and substantial amount of error which has accumulated in the data before these calculations). While condition numbers predict sensitivity to slight perturbation in the input, the exact perturbation may not be known and so the prediction can be overly pessimistic. Interval arithmetic is also pessimistic. When many calculations are performed, the intervals grow and special techniques must be used to narrow them. For geometric procedures, the operations on objects generally divide the objects into smaller sections. However, during this process the intervals widen, causing, in many instances, the intervals to become larger than the objects being manipulated.

3. The Pentagon Problem

Before attempting to fully analyze a proposed solution, it is necessary to precisely formulate a particular problem as a testbed for an accurate assessment of a possible approach. However, in most geometric algorithms the results of a computation are not known in advance and can only be checked by more numerical computations (making the testing suspect) or by viewing the results (making the viewer suspect). In our work, we have used the *pentagon problem* for experimentation.

The *pentagon problem* involves taking a pentagon stored as a set of five vertices (ten floating point numbers) and iterating *in* and *out* a certain number of times to get back to the original pentagon. An *in* iteration computes the intersection of the pentagon's diagonals resulting in a smaller inverted pentagon. This operation can be repeated on the ''new'' pentagon to get an even smaller pentagon. The inverse of this operation, the *out* iteration, projects alternate sides of the pentagon and finds the intersection point which is just a vertex of the larger pentagon (see Figures 4 and 5). In each iteration following the first, the data used are those calculated by the previous iteration. We assume that intermediate results are to be displayed (or used in some other way) so that multiple iterations can not be grouped to reduce error buildup. An iteration *in* and then *out* is an identity function, therefore, after an equal amount of *ins* and *outs* the differences between the

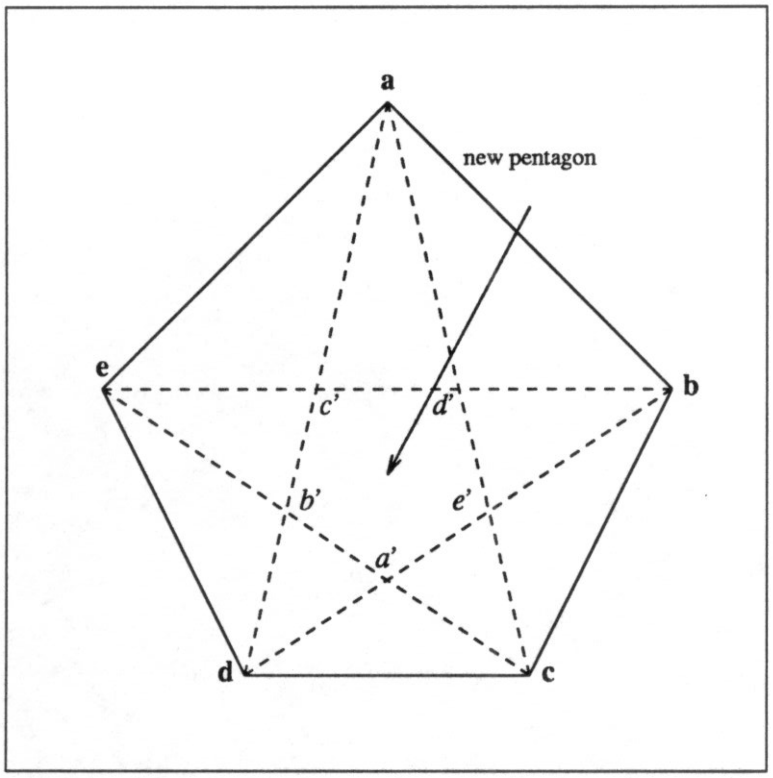


Figure 4: an *in* iteration

computed pentagon and the original pentagon can be determined. Owing to roundoff error in finite precision arithmetic, the computed vertices differ from the original vertices after a number of iterations *in* and *out*. If many iterations are performed, or the initial precision is too low, it may be impossible to maintain any accuracy in the calculated data.

The input to the *pentagon problem* consists of five vertices given as *x,y* coordinates. In computing the intersections for the *in* iterations, the diagonals are formed with the vertex of the original pentagon first and then with each subsequent pentagon. For example, referring to Figure

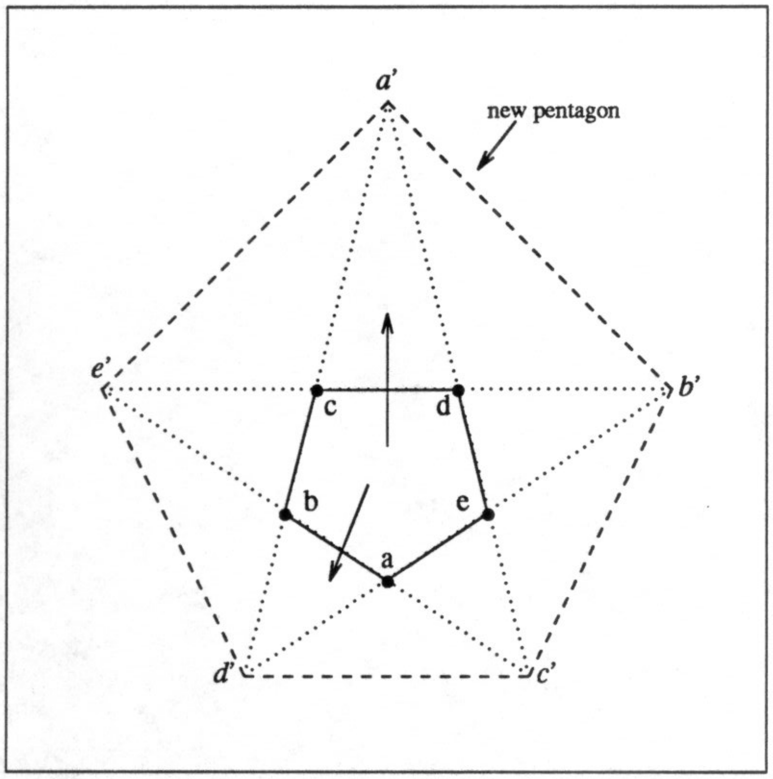


Figure 5: an *out* iteration

4, c' is the intersection of diagonals $\overline{ad}$ with $\overline{eb}$. Performing another iteration, c'' would be the intersection of $\overline{a'd'}$ with $\overline{e'b'}$. In the *out* iteration, the same line intersection function of the *in* iteration is used. Therefore, referring to Figure 5, c' is the intersection of $\overrightarrow{ba}$ with $\overrightarrow{de}$.

Note: While it is always possible to perform an *in* iteration first and then an *out* iteration to get back, it is not always possible to extend outwards starting with the initial pentagon. For example, if two sides of the original pentagon are parallel to each other, as in the case of a pentagon in the shape of a house, the parallel lines cannot be extended outward to intersect. There are other instances where it is possible to extend outwards with the original pentagon, but the result is not convex - e.g. if two semi-adjacent edges form angles of less than ninety degrees with the middle adjacent edge, as in a triangular shaped pentagon.

Although it is not a very practical problem, the *pentagon problem* captures the essence of cascaded intersections while enabling accurate testing of final and intermediate results. Furthermore, it has a simple structure and so it can be easily studied, yet it displays the unstable behavior of related (but more complex) iterative algorithms, especially those that are geometric in nature.

The key to the difficulty in the *pentagon problem*, as well as other cascaded geometric algorithms, lies in the fact that the entire set of iterations must be considered one unit, although the exact series of *ins* and *outs* may not be known in advance. Namely, an *accurate* assessment of *previous-error-generated* must be tracked and worked into the calculations to determine error accumulated at a particular level. Many of the mentioned methods aim towards the individual iterations and do not easily accommodate cascaded calculations. However, one method that does enable easy and accurate prediction of error generated during compounded calculations is the Permutation-Perturbation method of Vignes and La Porte [Vign74a].

Because the Permutation Perturbation method is stochastic it allows automatic error analysis (on-line), hence the number of exact significant decimal figures of any computed result can be obtained. The method performs its analysis by creating successive elements in the set of total possible computed results of an algorithm, until the number of significant figures in the result (average of the values) becomes stationary (refer to Chapter 2).

4. Application

The application of the Permutation-Perturbation method to the *pentagon problem* was straightforward. On each iteration, all intersection points were calculated three (four) [3] times using different permutations of the intersection code, and the number of significant digits of the average was computed with the formula of La Porte and Vignes (see end of this chapter). This was done for all five pentagon vertices (a total of ten floating point values - five x and five y) and the average of the number of significant digits of the vertices was plotted against the iteration number; the resulting curve represented the decline (or increase) of significant digits in the vertices as the iterations progressed (see the graphs at the end of this chapter).

(Note: A typical run of our hidden surface elimination program with 20,000 triangles involves over 200,000 iterations similar to those in the *pentagon problem.* [4] At each iteration as many as four new triangles can be created. Some of the triangles go through many more than ten such iterations, so the in-10 out-10 scheme is possibly overly conservative.)

Different combinations of iterations were performed: *in* ten then *out* ten; out-10 in-10; in-5 out-5 in-5 out-5; out-5 in-5; etc. [5]

Normally, the best computed result of an iteration was the average of the three calculated results of the different permutations of the intersection code [Mail79a]. However, this average value was not used as input data for all the intersection calculations of the next iteration. Namely, the three results of the previous iterations were stored and used in the next iteration (each dif-

[3] Samples are generated until the estimation of the number of significant digits stabilize. However, [Mail79a] has shown that three to four are generally enough.

[4] see next chapter and [Prog84a]

[5] It is not always possible to extend outwards starting with the initial pentagon as explained before.

ferent permutation used one of the results). This enabled the Permutation-Perturbation algorithm to artificially keep track of the calculations done thus far and use that ''knowledge'' in the significant digit calculation at any level.

This application of the Permutation Perturbation method is equivalent to just performing the entire sequence of calculations in advance using this method as a precision predictor for the complete set. The advantage of this method is that it can be stopped at any point in the series and the precision of the data at that moment can be computed (similar to interval arithmetic).

After all the iterations were completed, the final computed vertices were compared with the original vertices (which were given as input to the program) to determine the extent of the error incurred during the computations. The exact error was then compared to the computed predicted error to analyze the performance of the Permutation-Perturbation method. Fortunately, the round-off error buildup in the calculated results was within one digit of the actual error in all the tests performed.

Analysis of Results (for the in-10 out-10 series)

It is clear from the plots of significant digits vs. iteration number that the pentagons displayed similarities, thus certain conclusions could be drawn. All the curves were downward sloping, i.e. the number of significant digits in the calculations decreased as more calculations were performed on the data. Seen from a different perspective, the error increased as more computations were executed. The decline of significant digits (or increase in error) was slow at first, and then after a number of iterations (different for each pentagon) displayed loglinear (the significant digit is a log value) behavior (see the results of [Mara73a]). In general, the *out* iterations caused a steeper decline in the number of significant digits than the *in* iterations, mainly because the pentagon *grows* during the *out* iteration so that any error in the input data is magnified. Interestingly, when *in* iterations were performed after *out* iterations (for example, if the series was in-5 out-5 in-5 out-5) the plots exhibited a slight increase in the number of significant digits.

The iterations are rotation invariant -- a pentagon and its rotated version resulted in similar plots. Furthermore, the pentagons that were close to regular (equal angles) performed better than degenerate pentagons, implying that the shape of the pentagon was responsible for the curve as opposed to the actual coordinates of the vertices. The reason for these last observations is based on the fact that intersecting perpendicular lines gives a more accurate result than intersecting those that are close to parallel [Forr85a]. When intersecting ''almost'' parallel lines, the linear equations to be solved are nearly singular and the condition number is very high (slight perturbations in the input will cause large changes in the resulting output, see appendix for further explanation).

5. Other tests

Tests were also performed starting at different precision levels (the calculations were performed with higher precision). As can be seen from the graphs, the curves exhibited similar attributes to those performed at lower and higher precision values.

Other series combinations (in-5 out-5 etc.) displayed similar qualities to the in-10 out-10 series: the curve would decline during the *out* series and either stabilize or slightly improve initially during the *in* series (following *out* iterations) if the error generated previously was not overwhelming. For the in-1 out-1 in-1... series, the graphs displayed stable behavior with little error accumulation.

6. Analysis

Based upon the experimentation, the Permutation-Perturbation method proved to be helpful in predicting the amount of error accumulated during the cascading calculations of line intersections. Although it has an associated cost of a factor of three or four times that of doing no error calculation, it has many *significant* advantages to it over the other methods. First, it is mathematically easy to understand and implement (the code for this method is less than 50 lines of C, see appendix) which is no small achievement when dealing with numerical algorithms. It requires no special mathematical functions for the basic arithmetic operations and no extra hardware (which may or may not exist). Unlike the geometric flavored methods, no normalization or object rearranging is required. It can be implemented with any algorithm without modification to the method or recalculation of the mathematics involved (unlike condition numbers), and the method's calculations do not get messier as the program's computations progress. There are no special cases (such as division by zero in interval arithmetic) since the Permutation-Perturbation method is not interested in the individual computations. It is also an ''on-line'' algorithm and can be used during the programs normal run, not only as an error estimator but also to set the ''fuzz'' values in a program. Finally, this method provides an accurate estimate of error accumulation during the computations without being overly pessimistic or optimistic.

7. Permutation-Perturbation Method

7.1. Implementation

The mathematics of the Permutation-Perturbation method is explained in Chapter 2 and in the literature by Vignes and La Porte. The method is a general purpose solution and has been used to evaluate the local accuracy in the discrete Fourier transform [Bois80a]; for solving iterative optimization processes [Vign84a] and numerical intergration of ordinary differential equations [Alt83a]; and for stabilizing eigenvalue algorithms [Faye83a].

7.2. First Part

Fortran code that performs the Permutation-Perturbation method is given in [Vign78a] (it is approximately 50 lines). The program contains two parts. The first part randomly permutes the order of operations of three or more additions or subtractions. (Since subtraction creates the greatest error, in practice it suffices to apply permutations just to the addition and subtraction operations.) The computed values are also perturbed by toggling the least significant bit. This first part is run several times to obtain successive elements in the ''result'' population. In our implementation of the intersection algorithm for the *pentagon problem*, the permutations were precoded to speed up execution. Namely, different intersection algorithms were used to create the different results (e.g. slope method, s-t method, etc., see Appendix).

7.3. Second Part

The second part of the code performs the statistical analysis of the computed results and outputs the number of significant digits in the average. The formula used (from Chapter 2) is

$$C = \log_{10} \frac{|\bar{R}|}{\delta}$$

where $\bar{R}$ is the average of the computed results, and δ is the standard deviation (see [Mail79a] for information about the confidence interval).

7.4. Code

The code in pidgin C is as follows:

```
/* This procedure computes the number of significant digits of a set of computations. The
input consists of various results of different permutations and perturbations of the computations.
In [Vign78a] the code for this method in fortran is given (along with a permutation-perturbation
procedure). */

NCSE(r,N,c)

double r[]; /* contains the results from different permutations and perturbations of the input code (not
necessarily a double floating point value )*/
int N; /* the number of samples in r */
double *c; /* the results of ncse */

        {
        double avg,std;
        int i;

        /* first calculate the average and std. deviation */

        for (i=0,avg=0.0; i< N; i++)
                avg += r[i];
        avg /= (double) N;

        for (i = 0, std=0.0; i<N; i++)
                std += (r[i]-avg)**2; /* squared */
        std = sqrt(std/(double)(N-1));

        if (std==0.0) {
                *c=PRECISION; /* max precision being used */
                return;
                }

/* Care must be taken here if avg == 0 since the ''relative error'' of a number close to zero is undefined.
See [Vign87a] for more detail. */

        *c = log10(abs(avg/std));
        }
```

8. Graphs

8.1. Sample Plots and Discussion

The plots display executions of the pentagon problem with different parameters (iteration series). On the *y-axis* is the significant digit count (precision) predicted by the permutation perturbation method. The *x-axis* has the iteration count. On the abcissa, ticks going up signify an *in* iteration and ticks going down an *out* iteration. When the same number of *ins* and *outs* were performed (getting back to the original pentagon), the actual error was calculated using the formula

$$actual\_precision = \log_{10}(\frac{v_o}{|v_o - v_c|})$$

where v_o is the original vertex (inputted value), and v_c is the computed vertex. The predicted precision, as well as the actual precision (in italics), appears on the graphs. Below each of the graphs, there is a picture of the pentagon corresponding to that graph. The pentagons are drawn in a 30 by 30 box, and the vertices are labeled. The title of each plot contains the pentagon name (for reference in future plots).

8.2. Graphs and Pentagons:

a) Plots 1 through 4 are in-10 out-10 executions of the pentagon problem with different pentagons. The program was run with varying amounts of precision for the computations. There are some important points to notice in this set of graphs. First, the graphs are fairly linear after the first few iterations. Second, the more regular pentagons perform better (the end precision is higher than for the "not so regular" pentagons).

Furthermore, the overall shape of the graph does not change when using higher precision for the calculations.

b) Plots 5 and 6 display graphs of a pentagon and its rotation for the in-10 out-10 series. The graphs are almost identical in these cases.

c) Plot 7 shows an out-10 in-10 execution. When performing an *out* iteration on exact data (as is the case for beginning the series with an *out* iteration), the decline is much lower than the equivalent series starting with *in* iterations. One contributing factor is that division is generally not performed in this case.

d) Plots 8 and 9 are in-5 out-5 in-5 out-5. During the *in* iteration, following an *out* iteration, the precision increases slightly (this is only true if there is still enough precision left). However, if more *in* iterations are done as in Plot 10 (in-5 out-5 in-10 out-10), the precision will again start to decrease during the *in* iteration.

e) Plot 11 is a graph of an in-1 out-1 ... series. Again, during an *in* iteration following an *out* iteration, there is a slight improvement. In general, the overall decrease in precision for this case is very small.

8.3. Implementation Detail

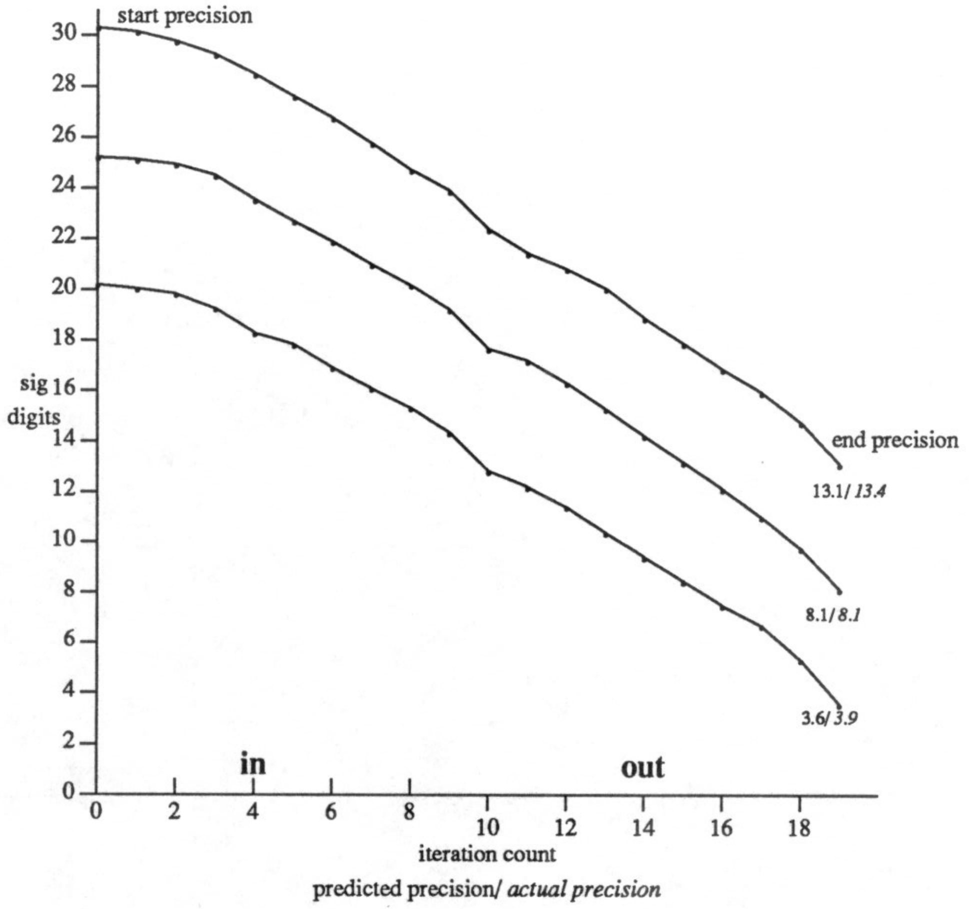
1. One of the first things to note is that precision cannot be created. If the input has only two significant digits, the output cannot really have more. However, the input can be thought of as having trailing zeros, thereby, ''increasing the precision'' of the input.

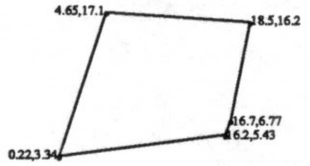
2. Another important point is the number of significant digits predicted by this method is not always analogous to the number of digits ''the same'' for two quantities since trailing nines and zero's are considered significant. For example, the number 1999 compared to the number 2000 has 3.3 significant digits (which is as it should be).

3. Below about two or three significant digits, the intersection could not be performed accurately for many instances. If the iterations are continued beyond this point, the results become meaningless as do the predicted number of significant digits in those results (the graph begins to rise and descend arbitrarily).

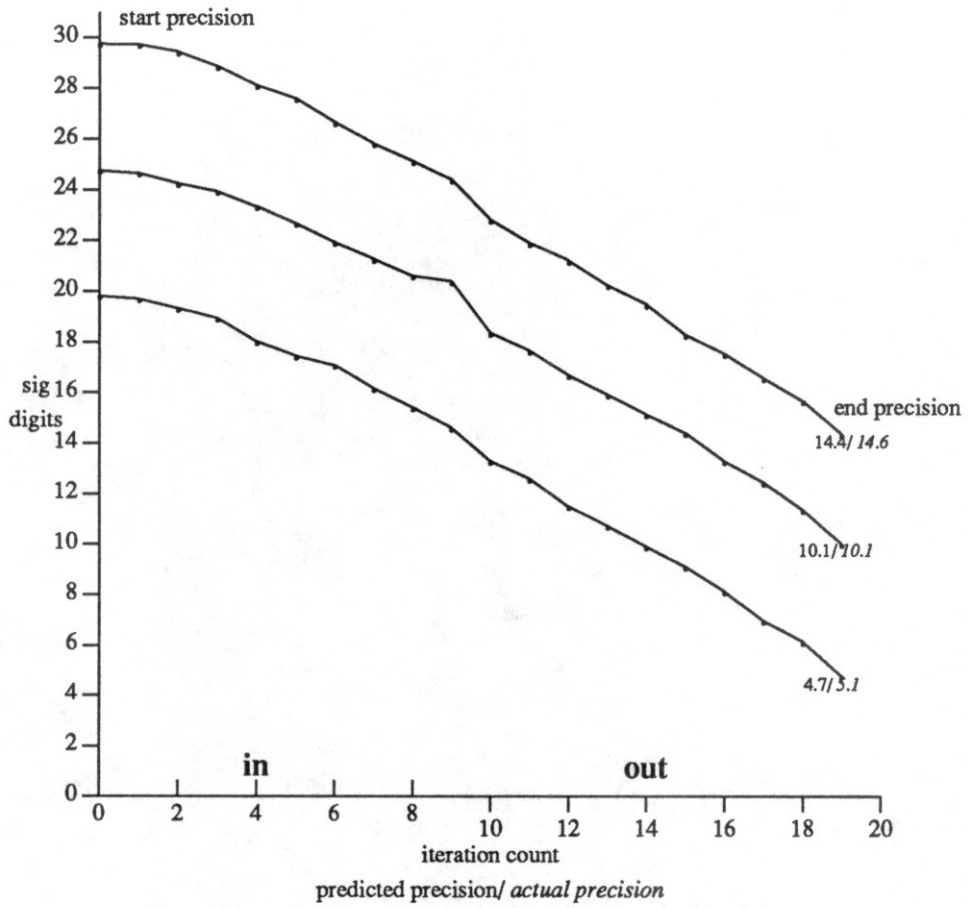
4. Care must be taken when performing the perturbations, since unstable computations may result, throwing the predicted accuracy off. In these cases, more samples must be taken.

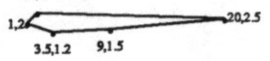
Plot 1: in-10 out-10 (pentagon 1)



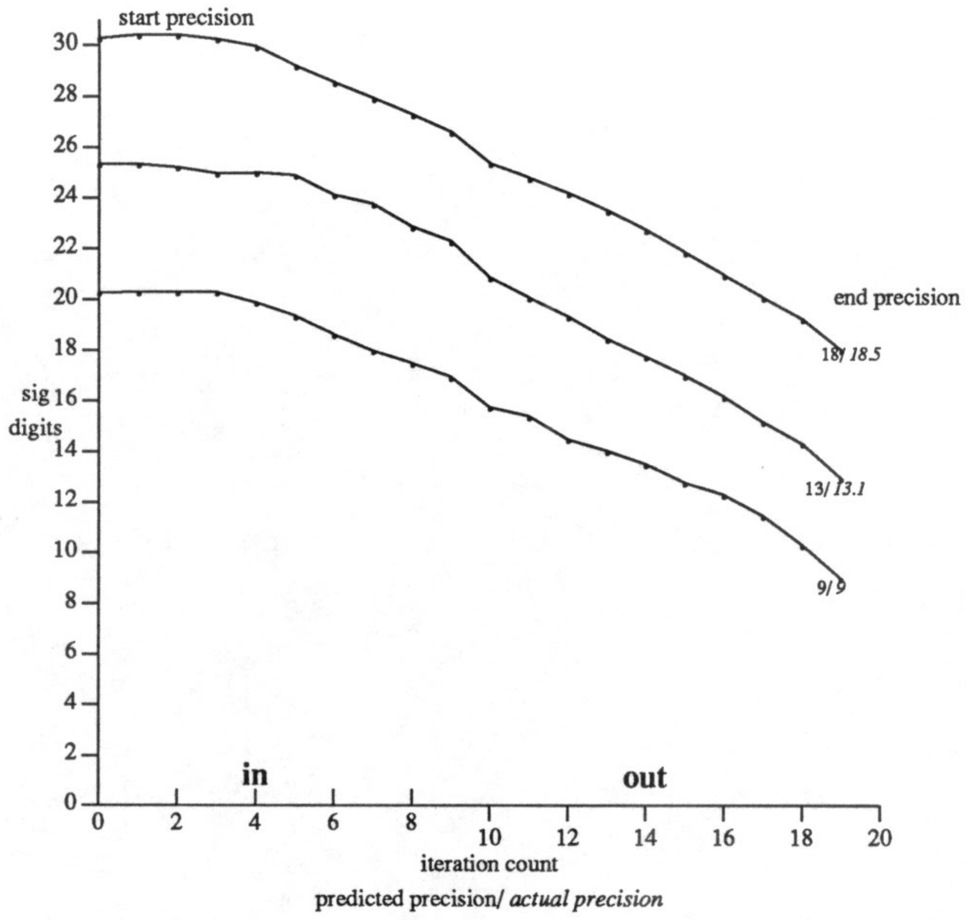


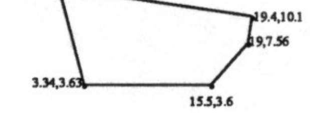
Plot 2: in-10 out-10 (pentagon 2)



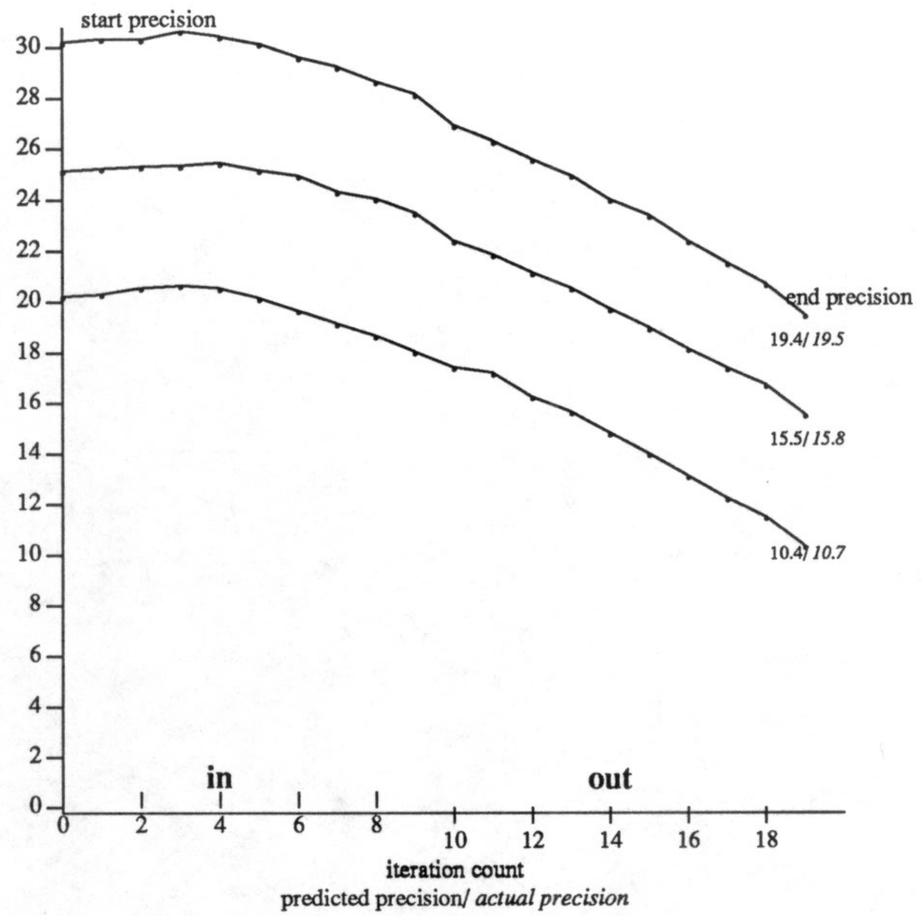


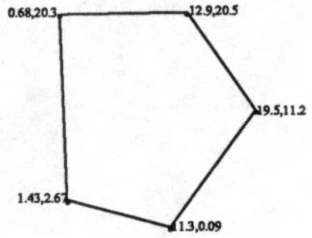
Plot 3: in-10 out-10 (pentagon 3)



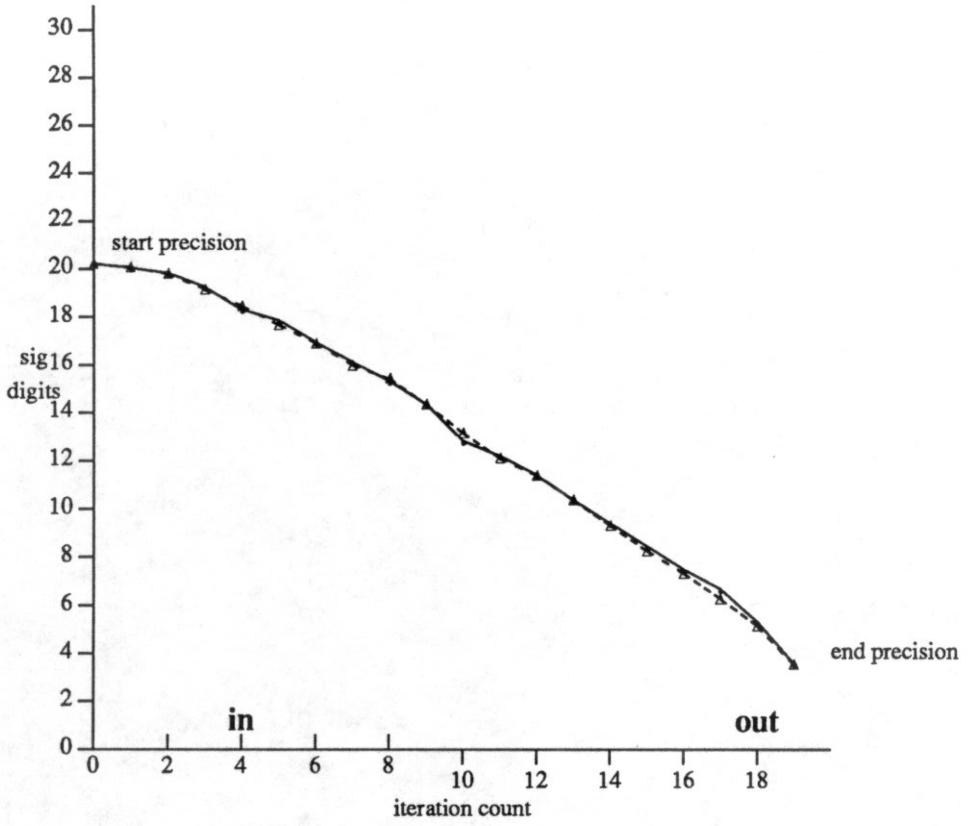


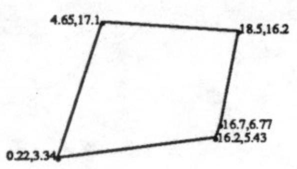
Plot 4: in-10 out-10 (pentagon 4)



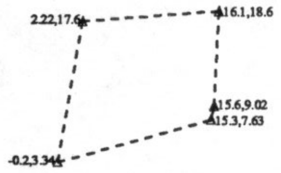


Plot 5: in-10 out-10 (pentagon 1)



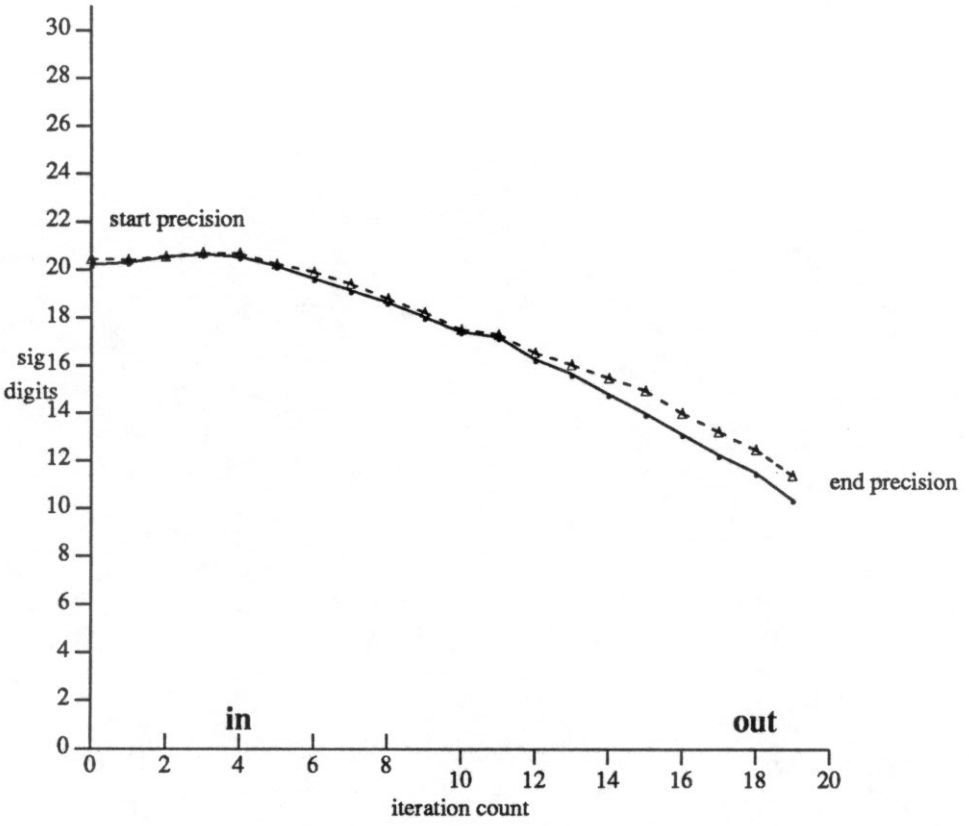


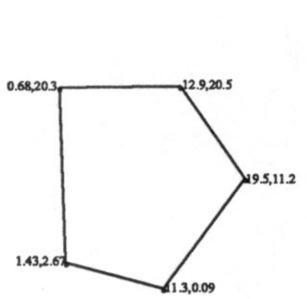
pentagon



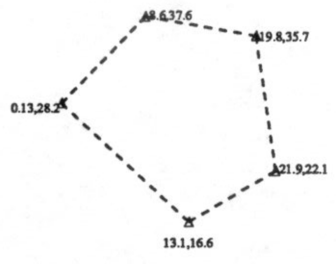
rotated and shifted pentagon

Plot 6: in-10 out-10 (pentagon 4)



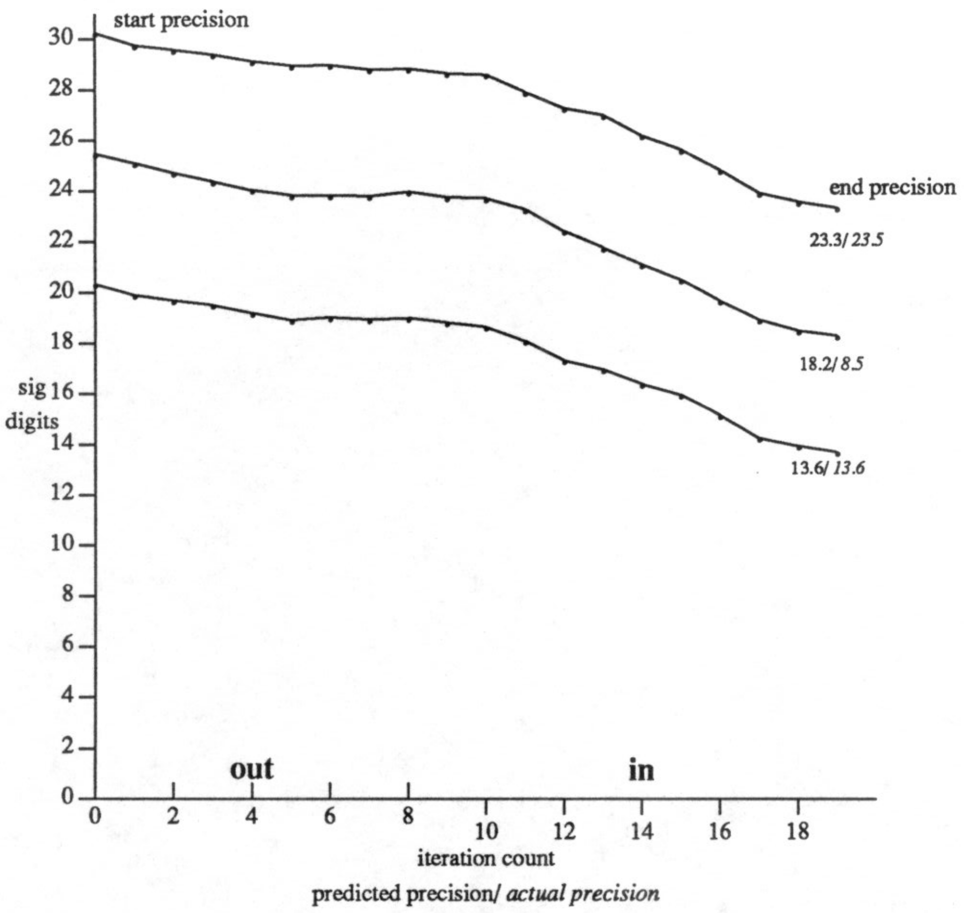


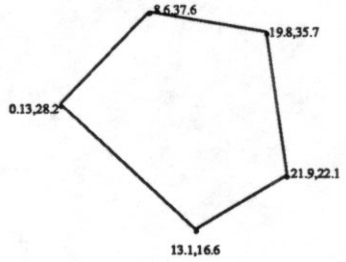
pentagon



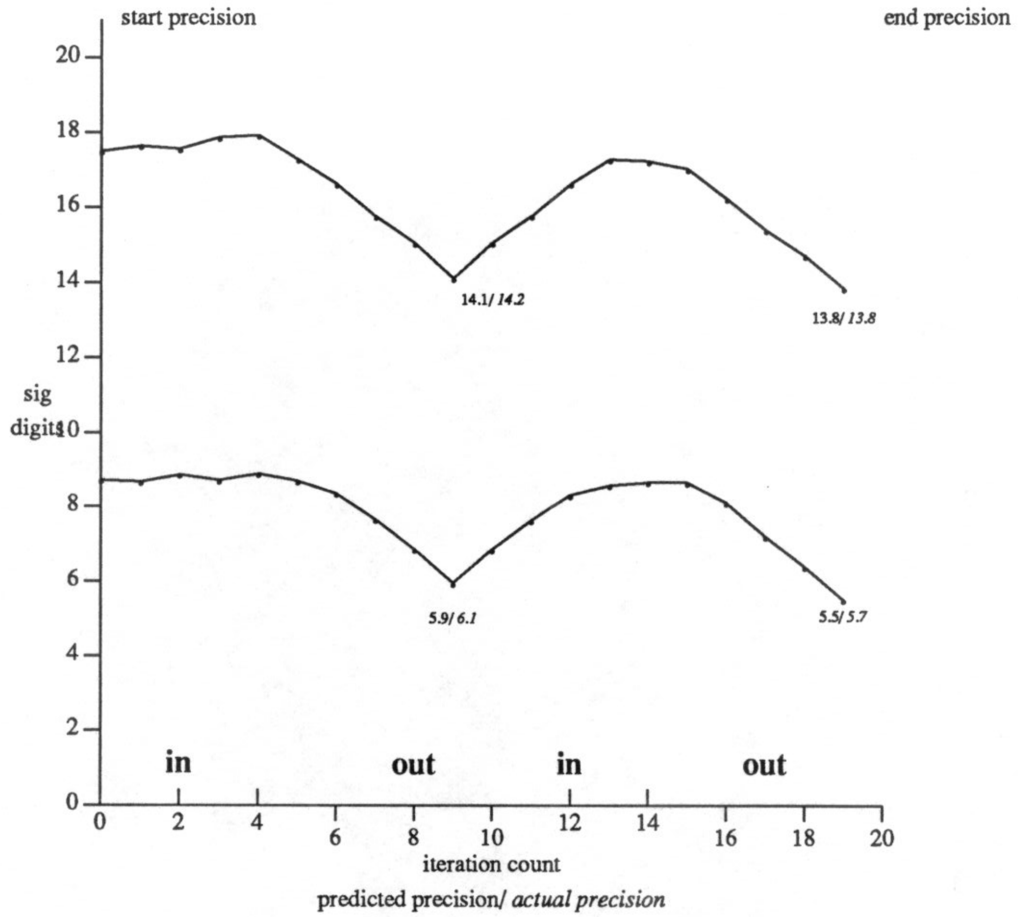
rotated and shifted pentagon

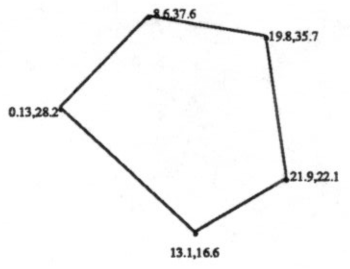
Plot 7: out-10 in-10 (pentagon 4)



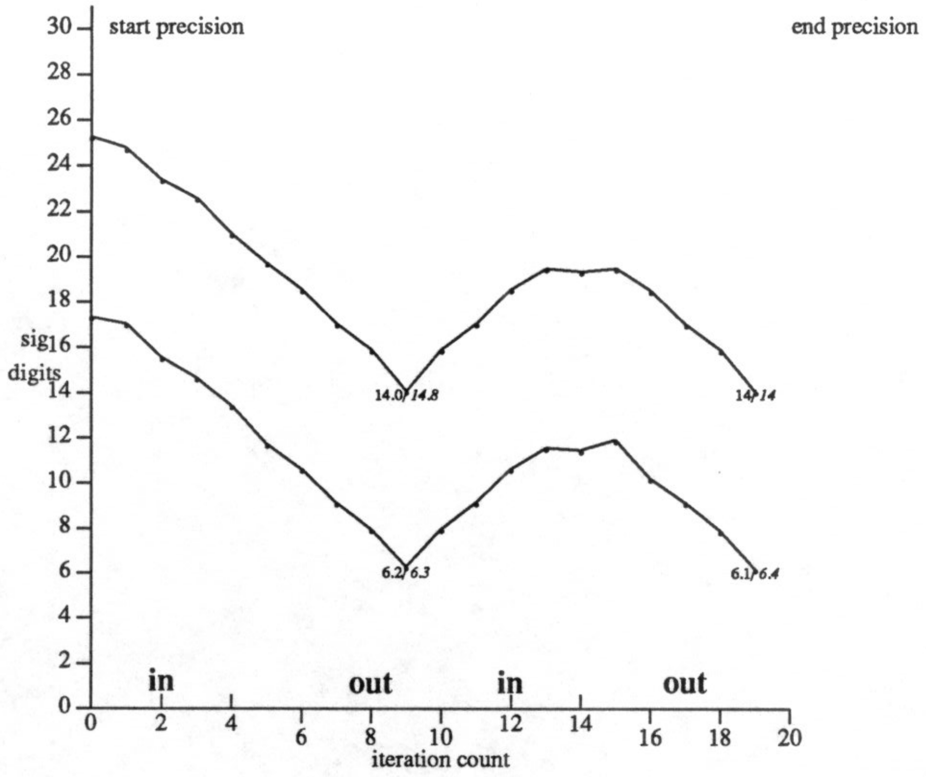


Plot 8: in-5 out-5 in-5 out-5 (pentagon 4)

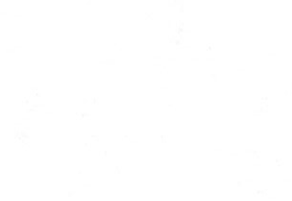




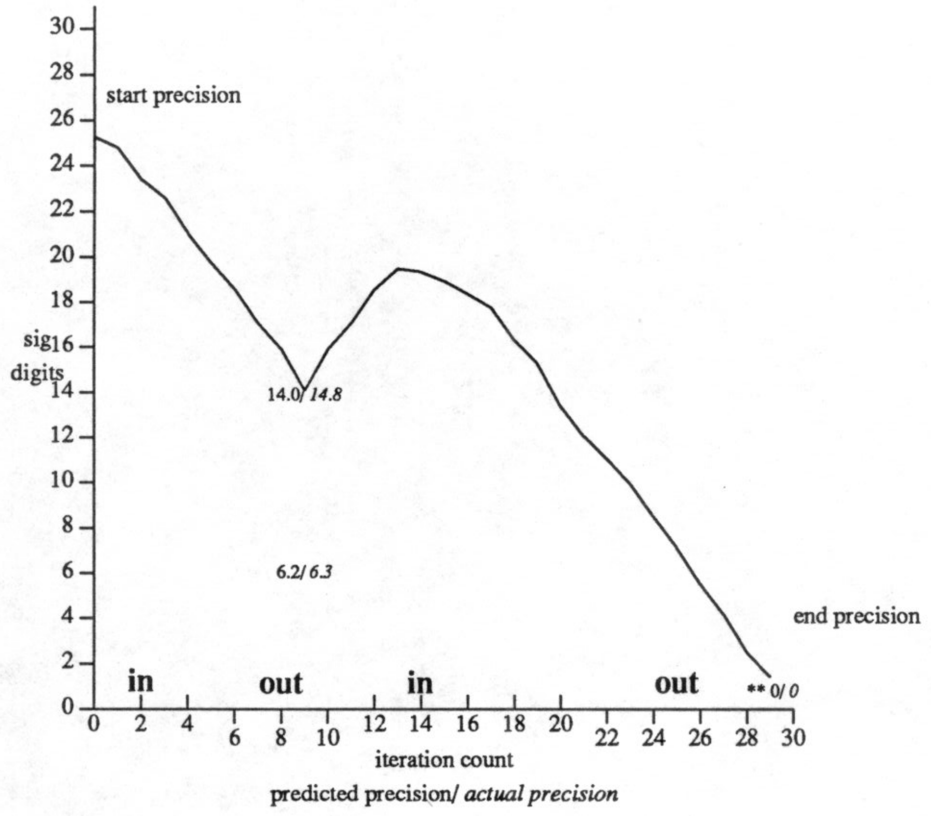
Plot 9: in-5 out-5 in-5 out-5 (pentagon 5)

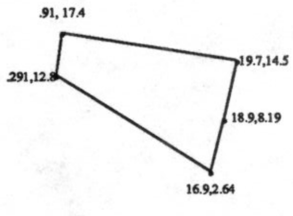


predicted precision/ *actual precision*

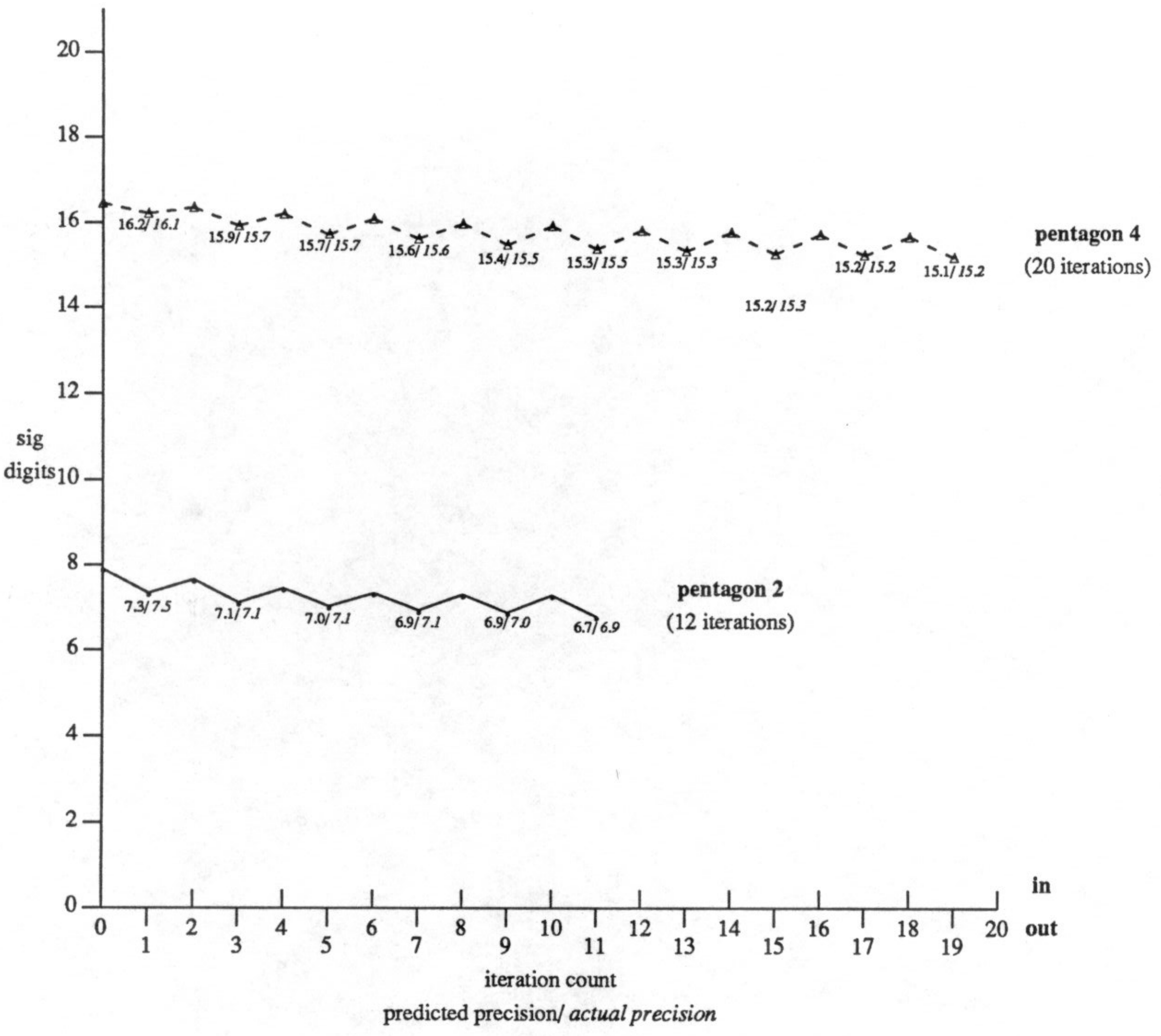


Plot 10: in-5 out-5 in-10 out-10 (pentagon 5)





Plot 11: in-1 out-1 ... (pentagons 2 and 4)



4

Extensions

Extensions

1. Problem

In this chapter a slightly different problem, which is an extension to the original problem, is considered:

> Suppose we are given a set of line segments along with a series of computations to be done on these segments. This computation will involve creating new segments having endpoints which are intersections of existing line segments. An additional part of the input is a specification of the precision to which the original inputs are known and the precision desired for the final output.

Our model of computation assumes that calculations can be done at any precision but there is a cost function dependent upon the precision of the computation. Furthermore, since the computation tree is known, backtracking is permitted in order to achieve greater precision. The cost of this backup is defined as the additional cost to redo the computations at the higher precision **added** to the cost of the computation already done. Finally, there is no advantage to achieving extra precision, however, a computation is deemed to be unacceptable if it does not achieve the desired precision. Basically, we envision three processes: one that does the actual calculations; a second to record the history of the computations; and a third to determine the precision and set the appropriate flags when necessary.

We claim that this is a valid model for hidden surface and many other computations in computer graphics. Indeed, our attention was focussed on this problem because of our frustration with *ad hoc* methods we were using to achieve desired precision in hidden surface routines we were writing as part of our graphics efforts. There are two versions of the problem stated above. In one, the entire computation tree is known in advance and for the other, the computation tree is determined as the computing evolves. In what follows, we focus on the first which is the simpler of the two.

Using the results of the previous chapter, the precision during the calculations and the amount of error can be computed easily. What is left now is to do something about the error. For example, if it is known that the computations do not have enough significant digits to produce the desired output, then the precision that the calculations were performed in should be increased somehow.

2. Multiple Precision

It is almost impossible to avoid increasing precision in order to boost accuracy. Assuming the cost of increased precision is somehow related to the amount of increase, one would like to avoid overkill, i.e. using much more precision than is actually necessary. If something is known about the accuracy of the data and the degradation of precision likely to occur with the computations to be performed, then hopefully that knowledge can help determine the precision to use. The section that follows discusses the issues involved in attempting to increase precision in conjunction with an accuracy measure. The increase in precision can be accomplished with any multiple precision package. Unfortunately, most are implemented in software and are therefore slow and cumbersome to use. [1]

3. Combining the two

The first problem that arises is merging the accuracy measure and multiple precision package. The tools must be put together in an efficient manner to produce a viable and effective combination. The most obvious (and costly) route to a workable mix is the following:

1. Estimate an initial precision;
2. At each step of the computation, determine the number of significant digits remaining;
3. If the number in step 2 becomes too low, increase the initial precision estimate and start again;

This algorithm works but raises more questions than it answers, such as, What should the initial precision be? What is ''too low''? And, how much precision is needed for the increase? etc... There is also potential for gross inefficiency. If the ''new'' precision in step 3. is inadequate, the whole algorithm must be repeated (over and over) until the correct precision is attained. Many of these problems are dependent upon the particular set of calculations and the computing environment being used for the implementation. Even for simple iterative procedures, these question remain.

Before proceeding, it is essential to further define two problem areas, namely, processes based on reusing computed data (i.e. cascading calculations) and cost functions. The simplest cascading process has three basic properties: the total number of iterations is known in advance; all the iterations accumulate error in a similar manner; and the sequence of calculations already

[1] In [Knot87a] code for a multiple precision program is given. See also [Schwa]

performed is available at little or no cost. Thus, the time/space tradeoff for recording the computation history to ease rollbacks can be ignored. In what follows, we assume that a rollback to a previous computation is always free. Note that the *pentagon problem* has these three qualities and is therefore an ideal model for initial experimentation.

The next problem concerns the multiple precision package. We would like to have a polynomial function $f(p)$ such that computations of precision p require $f(p)$ operations. In real environments, this is not necessarily valid. Computer hardware supports certain precisions (typically single and double, occasionally quad) for which computations are fast, while other calculations are done via software and are much slower. In this case $f(p)$ grows quadratically for small p, and decreases towards $O(p \log p)$ as p increases (not including the extra time needed for communication between software processes). To simplify our development, $f(p)$ is assumed to be either quadratic or linear.

With these issues clarified, we are now able to proceed with the implementation. Here again, numerous problems arise, which we state along with some of our empirical observations:

3.1. Problem 1:

How can the precision needed for future calculations be predicted? Some type of formula must be used to determine when, where and how to increase precision. The formula must account for the current precision, future computations, and future precision. If the rate of decline is stable and can be calculated during the computations, or if it is known in advance, then the number of significant digits of the final computed result can be estimated using the formula

$$final\_precision = a - i \times m$$

where a is the number of significant digits currently (in the execution), i is the number of iterations left to be performed, and m is the rate of decline per iteration. If the decline is linear then

$$m = b - a$$

where b is the number of significant digits before a particular iteration in which a digits remain after. Alternately, a larger ''history'' can be used to produce a more accurate m estimate, i.e. using the precision of the last two iterations instead of just that of the last one. The extended formula is more useful for cases which are not as stable and display slight change in the behavior of error accumulation (see the discussion of the *pentagon problem* later in this chapter).

3.2. Problem 2

Assuming the diagnosis of Problem 1 is accurate, it is necessary to provide a remedy or cure for the cases where insufficient precision for future calculations is predicted. There are two possibilities: backtracking and performing previous calculations with higher precision; or increasing the precision of calculations done from that point henceforth. The problem with both of these solutions are apparent. Do they work? If so, which one is preferable? And, how much precision is needed for the increase?

Based upon some initial experimentation with the *pentagon problem*, increasing precision for future calculations without backtracking was ineffective in most instances in stabilizing the significant digit decline (although it did slow down the rate of decline). Therefore, rolling back iterations was required. But major questions still remain unanswered, such as, how far to backtrack, and how much additional precision is necessary for redoing the calculations. Making the wrong decision has a serious effect on the performance of the system causing constant *zigzaging* or *thrashing*, i.e. repeated backtracking with increased precision until the correct amount is finally attained. An example of this is illustrated in Figure 1. The desired end precision for the execution (in-10 out-10) is 6 significant digits. Every time the program predicted that the current precision was not high enough to guarantee 6 digits after the complete twenty iterations, the precision was increased by two digits and backtracking was performed. As is evident from the graph, this is not the most efficient way to achieve the desired precision.

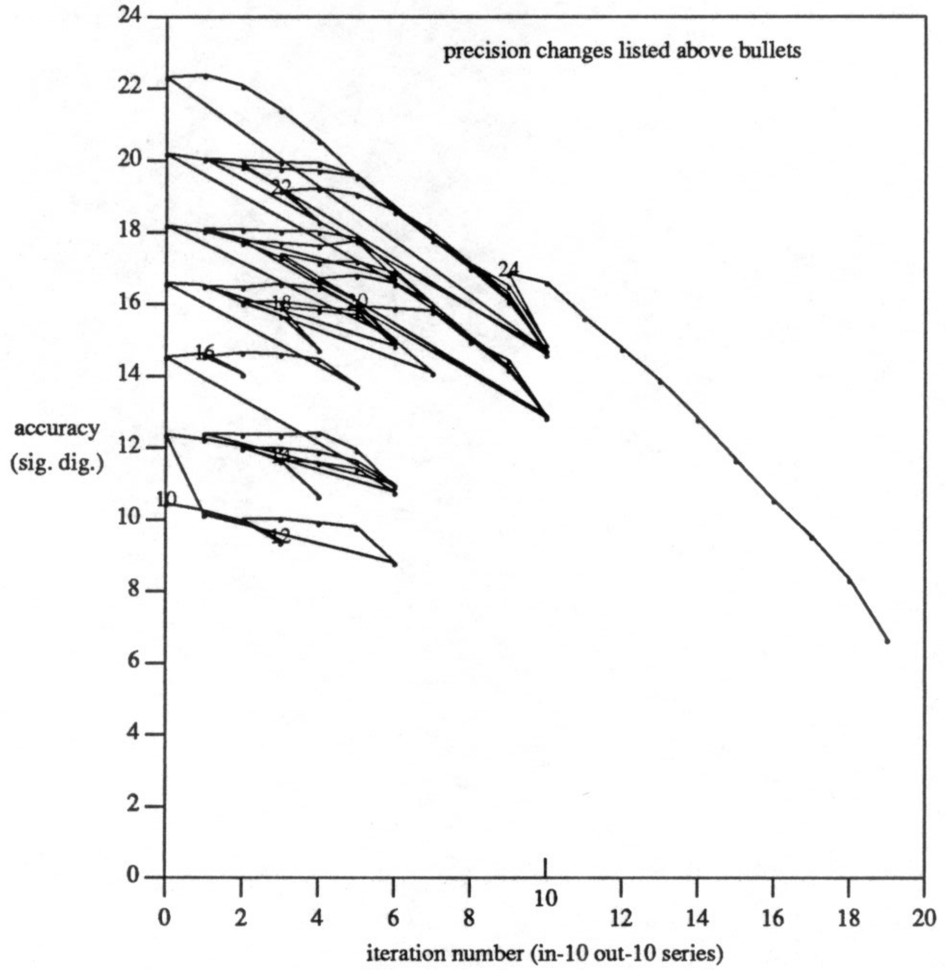


Figure 1: *zigzagging* **example**

In the implementation, both increasing precision with and without backtracking was performed. When increasing precision without backup (just redoing that particular iteration but not any previous ones) did not improve the significant digit count, then the program rolled back one iteration at a time increasing precision at each step until it was able to proceed with the calculations. However, finding the optimal amount of precision for the increase and an appropriate starting precision was very important, otherwise the cost became too high as a result of zigzagging and waste. Our empirical evidence seems to suggest that the addition of five to ten digits of precision works best (in cases where the desired final precision is 5-7 digits). This leads to the next major issue.

3.3. Problem 3

Ultimately, efficiency has to be a central component of the solution to this problem. The algorithm proposed for Problem 2 did not take efficiency into account. Therefore, it was sufficient to add precision and backtrack as often as was necessary. Ideally, backtracking costs should be taken into account. To do so, a formula is necessary which balances the cost of additional iterations at higher precision vs. the cost of overestimating the necessary precision. In our initial experiments, we observed no difference in comparative computational costs between cost functions which were linear and quadratic. Both suggested the same basic conclusion, namely, that backtracking should be avoided if possible, and thrashing should always be avoided.

Based upon the cost function, other questions arise. For example, to avoid zigzagging, it may be more beneficial to run the program in a ‘‘diagnostic’’ mode first and then run it again with the precision deemed necessary by the first run. This decision is heavily dependent on the cost function used. In particular, using a linear cost function for this two phase approach does not make sense, whereas for a squared cost function it becomes a more reasonable solution (if the precision to use for the second run can be determined). However, whether thrashing will occur with a particular set of input parameters (number of digits for the increase, the starting precision, etc.) cannot be known in advance.

Since the pentagon problem displays a linear degradation curve for the in-10 out-10 series, once the initial decline begins, the end precision can be calculated easily. Furthermore, if the precision calculated is deemed insufficient, a correct starting precision can be calculated and the program restarted. Namely,

$$starting\_precision = d + m \times i$$

where d is the desired end precision, m the rate of decline, and i the number of iterations left to be performed.

There are a number of advantages to executing in this manner. Because the decline begins relatively soon after the first couple iterations, only those need be repeated, thereby removing the additional expense of running the predictor method for the whole set of equations. In general, if the decline of precision can be caught early the cost of fixing it will probably be less. Furthermore, by restarting the entire sequence of calculations, a log or history file is not required to keep track of which calculations were already performed (or need repeating).

3.3.1. Average vs. Each

Another issue affecting both the cost and predictor formulas is that of average case vs. each case precision. It is unclear what result should be used for the ''current'' significant digit value (the a variable) in the formula to predict the precision of the final computed data. Should each computed value's significant digit count be used in the formula or should the average significant digit count of the computed values for a particular iteration be substituted? This may seem like a trivial problem, but it seriously affects the performance of the system. It is especially true for the *pentagon problem*, since basing the decision to increase precision on each value calculated (rather than the average precision of an entire iteration) resulted in much higher cost and much higher final precision. Whereas, while using the average number of significant digits of all the computed data in an iteration, costs were lower and the final precision was targeted more towards the ''desired'' precision. (The main reason for this is that most of the ''current'' data affects all the ''new'' data since four ''old'' vertices are required for the calculation of one ''new'' vertex.) However, although the average final precision may be equal to or above the ''desired'' precision, there can be individual values with less significant digits. For the *pentagon problem*, an end test was performed to verify that each final result's precision was higher than a lower bound threshold, and actions were taken if the values were lower (backtracking, etc..). The cost of implementing the lower-bound threshold was still less than testing each value (see the plots at the end of this chapter). But only applications in which the average precision of the system (at any time) is easily computed can avoid testing each value for significant digit decline. In the *pentagon problem*, each iteration is a closed unit making the average easy to determine, but for other applications, a distinct separation may not necessarily exist.

3.4. Problem 4

Lastly, there is the issue of generalizing our results for problems which are more complex than the *pentagon problem*. Unless we know otherwise, we can only assume that each iteration does computations with the same tendency towards roundoff errors. If the exact cascading process is unknown, we may want to run preliminary tests to gain some insight into the expected number of iterations that will be applied to individual data during the cascading process. Otherwise, every time the program is executed with a different set of data, an initial run at low precision can be performed to estimate this amount (in diagnostic mode).

For the *pentagon problem* the cascading sequence determined the behavior and characteristic of the corresponding precision plot. Namely, the in-10 out-10 series displayed linear decline and so the correct precision was easy to calculate (see above). For the in-out-in-out... series the graph would slightly improve during an *in* iteration following an *out* iteration. Therefore, a formula for predicting the final precision would have to be adjusted for this increase. However, since the characteristics of the general behavior of the pentagons is known for all these cases (most of the iteration sequences), it is easy to account for the cases in a program which performs the *pentagon problem.* Any problem can be studied in this manner (the Permutation-Perturbation method accommodates this easily) and the different cases noted.

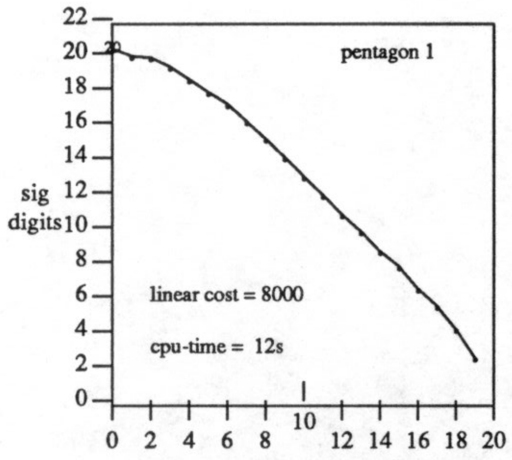
The final issue is the time/space tradeoff of storing the cascade if the intersection pattern is not known. At one extreme, we could store nothing and restart the process whenever precision gets too low. At the other extreme, the entire sequence of calculations can be recorded simplifying backtracking, as in the *pentagon problem* (although for the *pentagon problem* each iteration was essentially the same as the previous one). Further study is necessary to fully resolve these questions.

4. Leftovers

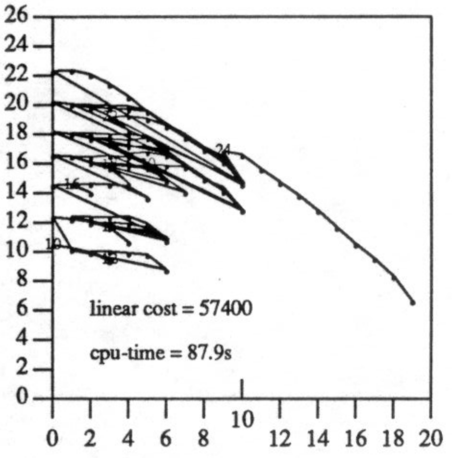
As is the nature of experimental work in computer science, as opposed to research done in more theoretical areas, different results are observed and, because of that, various solutions are proposed. The aim here has been to explore some of these differences on a basic problem of significant practical import and provide the groundwork for additional research in this area.

The initial results are promising, showing that more work needs to be done to precisely formulate the exact requirements of a system like this. The roundoff-estimator does work. Coupling this with the right ''guesswork'' as to when to increase precision and by how much leads to a robust solution for cascading line intersection, as well as for other more complex problems of computational geometry. The problem of completely removing the ''guesswork'' remains open. Unfortunately, the cost is high regardless of how it is measured, but this is only to be expected. As Demmel [Demm86a] observes, ''In short, there is no free lunch when trying to write reliable code.'' However, this solution seems to be less expensive than many others.

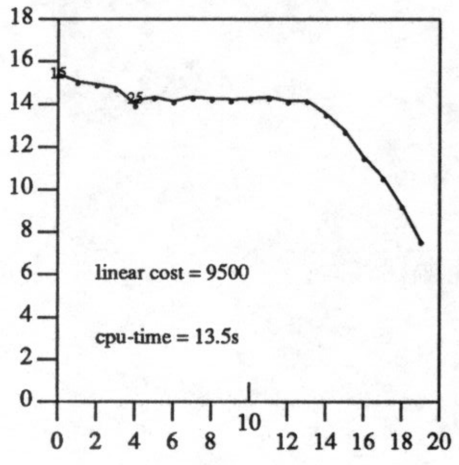
desired end precision (for all graphs) = 5



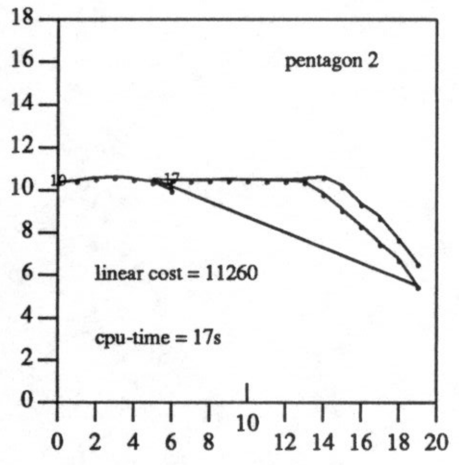
diagnostic mode
no precision increase was performed



starting precision = 10
add-on precision = 2



starting precision = 15
add-on precision = 10



starting precision = 10
add-on precision = 7
with a lower bound threshold = 4
rollback occurs if a value has fewer
significant digits than the threshold, even though the average is above

Cost Function Examples

5. Applications to Other Problems

5.1. Hidden Surface Elimination

A major advantage of the Permutation-Perturbation method is its portability. Because little analysis or programming is required, it can simply be implemented in conjunction with any numerical algorithm or procedure. Besides predicting the precision of computed results, it can also track the degradation of roundoff error (as in the *pentagon problem*) which can eventually lead to program failure.

One of the main difficulties in implementing many numerical processes with floating point arithmetic is that various *epsilon* or *fuzz* values must be set to compensate for inexact computations. The epsilon values appear throughout a program to guide decision making functions. Unfortunately, most of these epsilon values are set arbitrarily at the start of program execution and do not account for changes in the precision of the data that results from performing repeated calculations (e.g. cascaded calculations). If an accurate estimate of the data precision is available, the epsilon values can be set based on that precision and can be ''upgraded'' to reflect the change in the accuracy of the data.

An object space hidden surface elimination program (based on polygon subdivision and resulting in a set of unique non overlapping polygons) is a good example of this. Epsilon values help determine whether two lines intersect, whether vertices are coincident, when objects become degenerate, etc. In this case, the epsilon values are not necessarily related to the size of the ''original'' input, since the objects can be broken up into little pieces especially when they are partially obscured by other objects.

We have tested the Permutation-Perturbation method with an object space hidden surface elimination program, and some interesting points were observed. (For this experimentation, the object based hidden surface elimination program used is called *g* [Prog84a]. It is triangle based i.e all the surfaces are subdivided into triangles. The triangles are first depth sorted, and then the set of seen, unique, and non-overlapping triangles is found by traversing the sorted list and computing the triangle intersections. The intersection regions found are also triangulated.)

Most notable among the observations was that the precision degradation for the *pentagon problem* was much steeper than for the hidden surface elimination (HSE) program. The main reason for this is that the *pentagon problem* reuses computed data at a faster rate than does the HSE program. In the HSE program a new line (or polygon) contains part of the original polygon that was divided. Whereas in the *pentagon problem* both endpoints of a ''new'' line are ''freshly'' computed, only one endppoint may be ''fresh'' in the HSE program (less propagation error occurs). Furthermore, only one type of ''iteration'', the *in* iteration, is performed in the HSE program. Therefore, the case history is limited and the degradation uniform for most inputs. The only difference is that the exact number of iterations is not known. A deep picture containing a lot of overlap will need more iterations than one that is sparse.

However, a precision estimator is very useful in determining the values for the various epsilons throughout the program. For example, when determining if two vertices are the same, their difference is calculated and then compared to some ''fuzz'' value. If the precision of the vertices is known, it is much easier to set the epsilon values intelligently. In this way, the epsilon values can be set on-line and can change as the precision of the intermediate objects degrade.

Furthermore, this method is very useful as a debugging tool to track and identify numerical problems which cause program failures. Generally, one computation does not cause problems, but a series of calculation slowly lead to the buildup of catastrophic error. The Permutation-Perturbation method allows the tracking of this buildup enabling more detailed analyzation of the algorithm used.

5.2. Parallel Computations

As an added benefit, this method is easily parallelized since it involves repeating a computation a number of times (essentially doing the same thing three or more times). The parallelizing can take place at the calculation level or at the program level. The precision of each calculation can be computed after repeated performance of that calculation. Alternately, the entire program can be run a number of times, and stopped when an accurate precision is requested (at that point the data is collected and the Permutation-Perturbation method executed). Thus the cost of the method can be reduced to that of the ''do-nothing'' mode, or essentially the added precision verification is for free. This is not true for many of the other solutions which are not easily parallelizable and require hardware implementations to speed up the calculations (e.g. interval arithmetic).

5

Conclusion

Conclusion

1. History

This research grew out of an effort to produce a robust object space hidden surface elimination program. One of the main sources of program failure (and headaches) was the seemingly arbitrary settings of the epsilon values occurring throughout the program. The epsilon values were generally static values and could not account for the cascading calculations that contributed to the slow buildup of error. One set of calculations was generally not responsible, but a series of calculations would lead to catastrophic accumulation of roundoff error. The first issue that arose in trying to handle the epsilons in a more robust way (as opposed to the *ad-hoc* method being used) was determining the accuracy of the data being operated upon. We found that many of the classic numerical analysis techniques were too pessimistic or too difficult to implement effectively. The search for a simple and viable solution was initiated.

During the search, the general area of robustness (or lack of) in geometric processes was studied. It was evident that the need for awareness of many of the issues involved in producing robust and reliable procedures existed, and so our goals were expanded.

The aim of this research is to present and analyze systematic approaches to handle unreliability and enable more robust solutions to problems in applied computational geometry. In addition to exploring the issues involved, our goal is to provide a method to calculate and handle error in certain situations, and, of course, lay the groundwork for additional research. As mentioned in the first chapter, it would be nice to have a handbook for robust geometric computations. Hopefully, what has been presented here will become a piece of that unending and on-going effort.

The method we found most helpful was the Permutation-Perturbation approach. It is especially suited for cascading calculations and requires comparatively little effort to implement. Whereas the deterministic solutions for precision evaluation compute upper bounds on the round-off error, this method enables an *accurate assessment of accuracy*. Besides providing the user with some notion about the precision of a computed result and thereby granting more control over the output, many of the difficulties that arise from floating point arithmetic can be attacked once an accurate estimation of error is obtained, including the original motivation question of how to set the epsilon values. These quantities can be assigned on-line, relative to the accuracy of the data being computed. Other disorders leading to unreliability (unstable algorithms, etc.) can also be diagnosed and corrective measures taken. Furthermore, the method can be used as a debugging tool to track and find the problem areas which cause erroneous results and program failures.

2. Step Towards Robustness

We have presented in this research a **step towards robustness**. The results are as follows:

- We have explored and enumerated the issues involved in producing robust geometric procedures.
- The proposed solutions to this problem were explained, and a framework for them was given.

- We have singled out one of these methods and have shown how it can be used to estimate precision in an ongoing geometric computation.
- The method was then applied to a sample geometric process and its extension was discussed, thereby, presenting an empirical solution to a cascaded line intersection procedure.
- In addition, we have begun to investigate further uses and benefits of this solution.

So is it now time to throw away the *ad-hoc* working code? Not quite. The robust solutions are still too costly and inefficient. Needless to say, the work is far from complete. Robustness and reliability in computer science is an on-going issue. New technologies and new algorithms demand new attention to the never ending problem of error accumulation. It is unclear whether any procedure, geometric or otherwise, can be 100% reliable (especially something human engineered). However, this should not provide a deterrent for expending effort in search of solutions.

2.1. Open Problems

There are many open issues yet unresolved. Some of these include:

- applying the techniques presented here to more complex geometric procedures;
- analyzing robust methods for all the geometric and graphic primitive operations;
- studying the cost of these methods in depth;
- and developing new robust geometric algorithms.

3. Final Remarks

What we have done here is just the beginning. Awareness in the theoretical and practical communities of the issues involved is an important first step. Rediscovering some ''tried and true'' solutions in the literature, adapting them for new implementations, and creating new methods to cope with unreliability, has to be the way to attack these obstacles. We hope this work will provide insights into the problem and spur on additional research, which can only result in more robust and reliable algorithms for the future.

Appendix

Floating Point Arithmetic

1. Floating Point Computation

A floating point number is a number in the form of

$$\pm\beta^e \times .d_1 d_2 d_3 \cdots \qquad 0 \le d_i \le \beta \quad \text{for all } i$$

where β is the base or radix (commonly 2, 8, 10, 16), e the exponent, and $.d_1 d_2 \cdots$ the mantissa (a fraction base β). This number is equal to

$$d_1\beta^e + d_2\beta^{e-1} + d_3\beta^{e-2} \cdots$$

Because computers have only finite storage capacity, a floating point system implemented on a machine generally has limits on the maximum size of the mantissa and exponent. Therefore, a floating point number system is defined by the four parameters $F(\beta,t,m,M)$, where t is the precision of the mantissa $(.d_1 d_2 d_3 \cdots d_t)$, m is the minimum exponent, and M is the maximum exponent ($m \le e \le M$). This is referred to as a **t-digit, base β floating point number**. When $d_1 > 0$ the floating point number is said to be *normalized.* Clearly, not all real numbers can be represented by a t-digit (finite) floating point number system.

2. Properties of a Floating Point Number System

One of the most interesting properties of a t-digit floating point system is that the set of representable numbers are not distributed evenly between the highest and lowest possible value.

For example, in the system $F(2,3,-1,2)$ the numbers are distributed as follows:

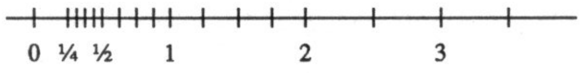


Fig 1. Positive numbers in *F(2,3,-1,2)*

From the graph it is evident that there are more numbers in the representable set between ¼ and 1, than between 1 and 2. Values which are not in this set must be approximated by the closest number in the set, therefore, a better approximation is attainable for values occurring in more "densely populated" intervals. This uneveness causes some of the complications which arise when analyzing numerical computations.

Because a floating point system is finite and discrete, there are important quantities associated with each system that can be calculated based on the parameters of the $F()$ function. Given a floating point system, $F(\beta,t,m,M)$ the smallest positive representable number, σ, is β^{m-1} and the largest, λ is $\beta^M(1-\beta^{-t})$. When $m \le e < M$ then there are $(\beta-1)\beta^{t-1}$ floating point numbers between β^{e-1} and β^e and these numbers are separated by a distance of β^{e-t}. One of the most important properties is the distance from 1 to the next larger floating point number. This is called the *machine epsilon*, ε, and is equal to β^{1-t}. In the example above, $F(2,3,-1,2)$, σ=¼, λ=3½, and ε=¼. For a VAX [a] format D_floating (double precision) 64 bits are used to represent a floating point number (8 for the exponent and 56 for the mantissa) with the parameters $F(2,56,127,127)$ the magnitude of (σ,λ) is between 0.29×10^{-38} and 1.7×10^{38}.

In general, the spacing between a floating point number x and its adjacent neighbor is at least $\frac{\varepsilon|x|}{\beta}$ and at most $\varepsilon|x|$ (except if x or the neighbor is 0). [1] **Note:** converting from one

[1] The spacing to the left and right of 1 are the extreme cases of relative spacing between nonzero consecutive numbers.

base to another is not exact. For example, 0.1 in base 10 (human language) is .0001100110011... in base 2 (computer language). However, any number with finitely many binary digits can be represented with finitely many decimal digits (if the number is *t* binary digits then it is at most *t* decimal digits).

3. Rounded/Chopped Numbers

As stated before, numbers which are not in the representable set for some floating point system must be approximated by one that is in the set. There are two basic methods which are used to do this approximation, *chopping* or *rounding*. If a number is truncated to *t-digits*, the precision of the system (by deleting the excess digits) this approximation is known as a *chopped value.* For a better approximation, the "true" value could be set to its closest representable number (member in the floating point set). This is called a *rounded value.* When the "true" value lies midway between two representable numbers it is rounded to the higher one (or the even one).

The absolute error associated with an approximated value (rounded or chopped) is just

$$E_a = |x - \hat{x}_a|$$

where x is the "true" value and $\hat{x}_a$ is the approximated one. The relative error is

$$\mathrm{E}_a = \frac{|x - \hat{x}_a|}{|x|} \quad \text{for } x \neq 0.$$

For a chopped value, a bound on the relative chopping error can be estimated by (for $x \neq 0$):

$$\frac{|x - \hat{x}_c|}{|x|} < \frac{\beta^{e-t}}{\beta^{e-1}} = \beta^{1-t}$$

The expression on the right, β^{1-t} is called the *unit chopping error* (it is also the *machine epsi-*

lon). For a rounded value, the bound is

$$\frac{|x-\hat{x}_r|}{|x|} < \frac{\beta^{e-t}}{2\beta^{e-1}} = \frac{1}{2}\beta^{1-t}$$

and the *unit rounding error* is $\frac{1}{2}\beta^{1-t}$. If a value is approximated to another, it is said to be correct to *N significant digits*, where N is the number digits which are the same in both values.

When comparing floating point numbers, it is sometimes important to discuss the interval distance between them - or places in the significant digits. This is the number of representable floating point numbers between the two values, and these intervals are referred to as *ulps* (*u*nits in the *l*ast *p*lace). For example, in $F(2,3,-1,2)$ (refer to Figure 1.), the numbers 1 and 2 differ by 3 *ulps*. Ulps are not necessarily integer values and can be used to express fractional intervals; between 2 and 2.75 there are 1.5 ulps.

4. Errors in Computer Arithmetic Operations

4.1. Relative vs. Absolute

In a finite floating point system, the computed value of some arithmetic operation will not necessarily be a member of the representable set and will contain error (either from rounding after the operation to fit the computer wordlength, or because of the limitations imposed by the computer arithmetic unit). This error can be calculated, as before, using an absolute or relative error measure. The absolute error is

$$E_{abs} = |x - x_c|$$

where x_c is the computed result, and x is the "true" value of the computations. A more useful quantity is the relative error measure since it gives a better sense of the magnitude of the error with respect to the "true" value, and it eliminates the dependence of the error on the size of the actual number. The relative error is

$$E_r = \frac{|x - x_c|}{|x|}.$$

It is sometimes convenient to express this quantity as

$$E_r = \frac{|x - x_c|}{|x_c|}$$

which is valid when x_c is a reasonable approximation to x (because it is only a relative measure). The term *roundoff error* is used to denote the error that occurs during the computation of arithmetic operations.

4.2. Computational Error

There are various types of errors (sometimes all gathered under the term roundoff error) which arise in computational processes:

a. actual roundoff error - the accumulation of error resulting from expressing unrepresentable numbers by representable ones (rounding or chopping). Also included are errors resulting from the limitations of the hardware in performing basic operations, and conversion error (converting from one base to another).

b. truncation error - If an infinite mathematical process (e.g. taylor series ...) is terminated after a finite number of steps, the resulting error is called truncation error.

c. propagation error. In addition to roundoff errors there is also the phenomena of error propagation, namely errors arising from evaluating functions with inexact or approximate data. The propagated error is

$$E_p = |f(X) - f(\hat{X})|$$

where $\hat{X}$ is some approximation to X. Note that this is much different than the error in a. which is $|x - \hat{x}|$.

d. human error (typos, erroneous equations)

These errors arise for the following basic reasons:

1. Precision of the available hardware: The larger the precision available for computation of mathematical functions the less the error accumulating will be noticeable on the results.
2. Number of operations being applied: Generally fewer operations will cause less growth of error.
3. Algorithm being used: Certain methods and solution to problems are ''better'' than others and tend to minimize the growth of roundoff error.

Of the three basic reasons, both 2. and 3. are in many instances, user-dependent, i.e. they are provided by the programmer, and are therefore the most likely targets in the study of roundoff error.

4.3. Error Factors

There are two basic inherent characteristics of a floating point number system which causes the errors listed above. The first factor provides the basis for much of the work in roundoff error analysis and it is that floating point arithmetic is *not associative*, namely

$$(A + B) + C \neq A + (B + C)$$

It is because of this that rearranging a set of calculations will result in different answers (some with less error). For example the four numbers

$$.1025\times10^{4}, \quad -.0123\times10^{3}, \quad -.9733\times10^{2}, \quad -.9315\times10^{1}.$$

when added left to right, using four digit arithmetic, results in the exact sum $.6755\times10^{1}$. However, adding from right to left produces $.7000\times10^{1}$. [Vand78a]

The second interesting property is subtractive ''catastrophic'' cancellation, i.e. the loss of significant digits through subtraction (of nearly equal numbers) and is a major source of error in numerical computations. For example, if

$$x = 12345678.0 \qquad \text{and} \qquad y = 12345677.0$$

their difference $(x-y)$ is 1.00000000. However, if x and y are slightly perturbed to

$$\hat{x} = 12345678.1 \qquad \text{and} \qquad \hat{y} = 12345676.9$$

their difference is now 1.20000000. Thus a change in the ninth figure of the original data has caused a change in the second figure of the answer [Vand78a]. In the example of non-associativity above, the wrong answer computed in the addition of the number from right to left was also caused by subtractive cancellation.

When discussing errors which arise in computational procedures, it is important to distinguish between methods or algorithms which cause more error to accumulate, and data which is inherently hard to calculate with. *Unstable methods* refer to methods or algorithms which produce poor results because of the way the data is handled. A typical example used to illustrate this is the formula for solving quadratic equations (which arises in many geometric calculations). The quadratic equation:

$$ax^2 + bx + c = 0 \qquad a \neq 0$$

has at most two real solutions given by

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

If the determinant ($\sqrt{b^2-4ac}$) is very close to b, (this can occur when $b^2 >> 4ac$) then a loss of significant digits will result because of subtractive cancellation (for the solution involving the addition operation). By transforming the equation, the subtraction can be avoided when $b>0$ with

$$x = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

An *unstable problem* or an *ill conditioned problem* refers to the sensitivity of the solution to slight changes in the input data (regardless of the method used to solve the problem). For example, the set of linear equations:

$$x + 2y = 3 \quad \text{and} \quad .499x + 1.001y = 1.5$$

has the solution $x=y=1.0$. However, if the second equation is changed to

$$.5x + 1.001y = 1.5$$

then the solution changes to $x=3$ and $y=0$. (In general, near singular equations are ill conditioned.) [Vand78a] When analyzing methods for error accumulation, care must be taken to distinguish between methods which are unstable vs. the problems which are unstable. One of the first steps in error analysis is testing the method or algorithms being implemented for instability. However, empirical testing is sometimes not a good indication of a ''bad'' method since the data may be ill-conditioned.

The two most common (and oldest) approaches to this problem are called *forward error analysis* and *backward error analysis*. *Forward error analysis* of an algorithm (or method to solve a problem) compares the exact solution of a problem with the computed solution. Unfortunately, an analysis of this type may be misleading if the problem is unstable. *Backward error analysis* is independent of the problem and so will give an indication of the stability of the method. The basis for this analysis is to consider the computed solution as the exact solution of a perturbed problem. If the perturbation is ''small'' then the algorithm is considered stable. For an unstable algorithm, the computed solution is the exact solution of a perturbed problem, where the perturbation is large compared to the original problem (as in the example above).

5. Error Estimation and Reduction

The two methods for error estimation are *a priori analyses* and *a postiori estimates*. *A priori analysis* is done before the calculations are actually carried out, and *a postiori estimates* are estimates done after the calculations. In the latter case, the computed value is used to estimate its own accuracy (forward and backward error analysis are of the former). Both of these methods can be used to either compute the error, detect unstable methods, or detect ill-conditioned problem sets and produce warnings. For error reduction, either precision is increased or alternate arithmetic and extended floating point systems are used.

Lines

1. The last word on lines

1.1. Lines and Line Segments in R^2

The classical formula for a line in R^2 is

$$\{(x,y) \mid y = mx + b\},$$

where m is the slope and b is the y-intercept (providing the line is not parallel to the y axis). The more general formula is

$$\{(x,y) \mid ax + by + c = 0\}.$$

If (x_1,y_1) and (x_2,y_2) are two points, then the slope of the line containing those points is $(y_2-y_1)/(x_2-x_1)$ and the y-intercept can be found by substituting any one of the two points into the line equation. For the triple (a,b,c) in the more general case, if $x_1=x_2$ (the line is parallel to the y axis) the triple is $(1,0,-x_1)$ (this forms the equation $x=c$ for some constant c), otherwise $((y_2-y_1)/(x_2-x_1), -1, y_i-x_i(y_2-y_1)/(x_2-x_1))$ for $i=1$ *or* 2. A line can also be viewed parametrically as going from $P_1=(x_1,y_1)$ to $P_2=(x_2,y_2)$ by the equation:

$$f(p) = P_1 + (P_2-P_1)t \qquad -\infty \le t \le \infty$$

For the line segment from P_1 to P_2 (inclusive), t takes on the values $0 \le t \le 1$. In this case, a point on the line segment lies a fraction t of the way from P_1 to P_2. For values "before" the line segment $t<0$, and for values "after" the line segment $t>1$. For the values P_1 and P_2, t is equal to 0 and 1 respectively.

A very useful formula pertaining to lines is that of the *signed area* between a point and a line. Given a point $P_a=(x_a,y_a)$ and a line containing the points $P_1=(x_1,y_1)$ and $P_2=(x_2,y_2)$ the determinant, D_{a12}, equals:

$$\begin{vmatrix} x_a & y_a & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}$$

The magnitude of this value (D_{a12}) is equal to twice the area of the triangle $\Delta_{P_aP_1P_2}$, and the sign of D_{a12} determines which half space the point lies in with respect to the line. For example, if P_b and P_a lie on opposites sides of the line $\overline{P_1P_2}$ then $sign(D_{a12}) \neq sign(D_{b12})$. If P_a lies on the line, then $D_{a12}=0$. A more compact way of writing the determinant is

$$D_{a12} = (x_a-x_1)(y_1-y_2) + (y_a-y_1)(x_2-x_1).$$

1.2. Line segment intersection

Given two lines defined by:

$$L_{12}\text{: } [P_1,P_2] \qquad \text{and} \qquad L_{ab}\text{: } [P_a,P_b]$$

with their corresponding equations :

$$L_{12} : ax+by+c=0 \qquad L_{ab} : dx+ey+f=0$$

or more formally (in the form $Ax = b$):

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -c \\ -f \end{bmatrix}$$

their intersection is at $x = ((bf - ec)/(ae - db))$ and $y = ((cd - fa)/(ae - db))$, unless the two lines are parallel, in which case $ae = db$ (the slopes are equal). For the parametric case,

$$L_{12} : P_1 + (P_2 - P_1)t \qquad L_{ab} : P_a + (P_b - P_a)s$$

the intersection point of the two lines is where the two equations are equal, i.e. when

$$P_1 + (P_2 - P_1)t = P_a + (P_b - P_a)s$$

Solving for either s or t and then substituting this value back into the equation will yield the x,y value of the intersection point. Setting the x and y parts equal to each other yields

$$x_1 + (x_2 - x_1)t = x_a + (x_b - x_a)s \quad \text{and} \quad y_1 + (y_2 - y_1)t = y_a + (y_b - y_a)s$$

or

$$\begin{bmatrix} x_2 - x_1 & x_a - x_b \\ y_2 - y_1 & y_a - y_b \end{bmatrix} \begin{bmatrix} t \\ s \end{bmatrix} = \begin{bmatrix} x_a - x_1 \\ y_a - y_1 \end{bmatrix}$$

These are just two linear equations with two unknowns which can be solved easily:

$$s = \frac{((x_1 - x_a)(y_a - y_b) + (y_a - y_1)(x_b - x_a))}{((x_1 - x_2)(y_a - y_b) - (y_1 - y_2)(x_a - x_b))} \qquad \text{and}$$

$$t = \frac{((x_1 - x_a)(y_1 - y_2) + (y_a - y_1)(x_1 - x_2))}{((x_1 - x_2)(y_a - y_b) - (y_1 - y_2)(x_a - x_b))}.$$

Alternately, the expressions can be viewed as:

$$s = \frac{D_{1ab}}{D_{1ab} - D_{2ab}} \qquad \text{and} \qquad t = \frac{D_{a12}}{D_{a12} - D_{b12}}$$

When the lines are parallel $s = \pm\infty$ and $t = \pm\infty$ (the denominator is equal to zero). Otherwise, the variables s and t determine where on the line the point of intersection lies. If the two lines are coincident then $s,t = 0/0$ since all the determinants in this case will equal zero. When both $0 \le s,t \le 1$ then the intersection point lies between $\overline{P_1P_2}$ and $\overline{P_aP_b}$ (on the line segments).

In both cases, using the triple (a,b,c) or the parametric equation, the final computed value (factored) is:

$$x_i = \frac{-(x_1y_2x_b - x_1y_2x_a - x_2x_ay_b - x_2y_1x_b + x_2y_1x_a + x_2y_aa_2 + x_1x_ay_b - x_1y_ax_b)}{(x_1y_a - x_1y_b + y_1x_b - y_1x_a - x_2y_a + x_2y_b - y_2x_b + y_2x_a)}$$
$$y_i = \frac{(-y_1x_2y_a + y_1x_2y_b + y_2x_1y_a - y_2x_1y_b + y_2x_ay_b - y_2y_ax_b - y_1x_ay_b + y_1y_ax_b)}{(x_1y_a - x_1y_b + y_1x_b - y_1x_a - x_2y_a + x_2y_b - y_2x_b + y_2x_a)}$$

Obviously, there are many different permutations and ways to evaluate these equations.

1.3. Line segment intersection detection

There are two methods to determine whether two line segments intersect.

1. s and t can be calculated and if they lie between 0 and 1 inclusive then the lines segments intersect.

2. If $sign(D_{a12}) \neq sign(D_{b12})$ and $sign(D_{1ab}) \neq sign(D_{2ab})$

Although care must be taken in both cases when the lines segments are coincident ($D_{a12} = D_{b12} = D_{1ab} = D_{2ab} = 0$). Because of floating point arithmetic, the values calculated are not tested against 0 (or 1) but a number "close" to zero or one.

2. Condition Numbers

The *condition number* of a set of computations measures the sensitivity of the equation to small perturbations in the input. If it is high, then the the system is labeled *badly conditioned* or *ill conditioned.* For finding the intersection point of two lines (solving the linear equations which involves inverting the matrix), the condition number is

$$\kappa = \|A\| \cdot \|A^{-1}\|$$

where A is the matrix formed with the coefficients of the linear equations, and $\|A\|$ is the norm (1,2, or ∞). The condition number of A can also be expressed in terms of the angle, β, between the two lines [Hoff88a] (assume that one of the lines has a 45° slope):

$$\kappa(\beta) = \frac{1}{sin(\beta/2)}$$

A nearly singular matrix will have a very high condition number. Similarly for lines separated by very small angles (nearly parallel) since then $\sin(\beta/2) \approx \beta/2$.

References

a. *VAX Architecture Handbook,* Digital Equipment Corporation, Maynard, MA.

Alt83a.

Alt, R., ''Minimizing the Error Propagation in the Integration of Ordinary Differential Equations,'' *Scientific Computing*, IMACS/North-Holland Publishing Company, 1983.

Boeh86a.

Boehm, H., R. Cartwright, M. Riggle, and M. O'Donnell, ''Exact Real Arithmetic: A Case Study in Higher Order Programming,'' *1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Mass., 1986.

Bois80a.

Bois, P. and J. Vignes, ''A Software for Evaluating Local Accuracy in the Fourier Transforms,'' *Mathematics and Computers in Simulation XXII*, pp. 141-150, 1980.

Bres87a.

Bresenham, J. E., ''Anomalies in Incremental Line Rastering,'' *Proceedings of NATO ASI on TFCG & CAD*, July 1987.

Buch88a.

Buchberger, B., ''Algebraic Mehtods for Non-Linear Computational Geometry,'' *Proceedings of the ACM Symposium on Computational Geometry*, June 1988.

Cort87a.

Corthout, M. and H. Jonkers, ''A New Point Containment Algorithm For B_Regions in the Discrete Plane,'' *Proceedings of the NATO Advanced Study Institute on Theoretical Foundations of Computer Graphics and CAD*, Il Ciocco, Italy, July 1987.

Demm86a.

Demmel, J., ''On Error Analysis in Arithmetic With Varying Relative Precision,'' TR 251, NYU Dept. of Computer Science, October 1986.

Dobk88a.

Dobkin, D. and D. Silver, ''Recipes for Geometry and Numerical Analysis; Part I: An Empirical Study,'' *Proceeding of the ACM Symposium on Computational Geometry*, June 1988.

Edel88a.

Edelsbrunner, H. and E. P. Mucke, ''Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms,'' *Proceedings of the ACM Symposium on Computational Geometry*, June 1988.

Eric88a.

Ericson, L. and C. Yap, ''The Design of LINETOOL, a Geometric Editor,'' *Proceedings of the ACM Symposium on Computational Geometry*, June 1988.

Faye83a.

Faye, J. P., ''Stabilizing Eigenvalue Algorithms,'' *Scientific Computing*, IMACS/North-Holland Publishing Company, 1983.

Faye85a.

Faye, J-P. and J. Vignes, ''Stochastic Approach Of The Permutation-Perturbation Method For Round-Off Error Analysis,'' *Applied Numerical Mathematics 1*, pp. 349-362, 1985.

Forr85a.

Forrest, A. R., ''Computational Geometry in Practice,'' in *Fundamental Algorithms for Computer Graphics*, ed. R. A. Earnshaw, Springer-Verlag , 1985.

Forr87a.

Forrest, A. R., ''Geometric Computing Environments: Computational Geometry Meets Software Engineering.,'' *Proceedings of the NATO Advanced Study Institute on TFCG & CAD*, p. Il Cioceo, Italy, July 1987.

Gree86a.

Greene, D. and F. Yao, ''Finite Resolution Computational Geometry,'' *27th Annual FOCS Conference Proceedings*, pp. 143-152, Oct. 1986.

Guib87a.

Guibas, L. and J. Stolfi, ''Ruler, Compass and Computer - The Design and Analysis of Geometric Algorithms,'' *Proceedings of NATO ASI on TFCG & CAD*, July 1987.

Hoff88b.

Hoffmann, C., ''The Problem of Accuracy and Robustness in Geometric Computation,'' Tech. Report CSD-TR-771 (CAPO Report CER-87-24), Computer Science Dept. Purdue University, April 1988.

Hoff88a.

Hoffmann, C., J. Hopcroft, and M. Karasick, ''Towards Implementing Robust Geometric Computations,'' *Proceedings of the ACM Symposium on Computational Geometry*, June 1988.

Horv74a.

Horvath, E., ''Some efficient stable sorting algorithms,'' *Proc. 6th Annual ACM Symposium on Theory of Computing*, pp. 194-215, 1974.

Kara88a.

Karasick, M., ''On the Representation and Manipulation of Rigid Solids,'' *Ph.D. dissertation*, Dept. of Computer Science, McGill University, Montreal, August 1988.

Knot87a.

Knott, G. and E. Jou, ''A Program to Determine Whether Two Line Segments Intersect,'' Technical Report CAR-TR-306, CS-TR-1884,DCR-86-05557, Computer Science Dept. University of Maryland at College Park, August 1987.

Kuli81a.

Kulisch, U. W. and W. L. Miranker, *Computer Arithmetic in Theory and Practice,* Academic Press, New York, NY, 1981.

Kuli83a.

Kulisch, U. W. and W. L. Miranker (eds.), *A New Approach to Scientific Computation,* Academic Press, Orlando, Fl., 1983. Proceedings of the Symposiun on A New Approach to Scientific Computation Sponsored by IBM and held at the IBM T. J. Watson Research Center on Aug 3, 1982.

Kuli86a.

Kulisch, U. W. and W. L. Miranker, "The Arithmetic of the Digital Computer: A New Approach,'' *SIAM Review*, vol. 28, no. 1, pp. 1-40, March 1986.

Madu84a.

Madur, S. and P. Koparkar, ''Interval Methods for Processing Geometric Objects,'' *IEEE Computer Graphics and Application*, pp. 7-17, February 1984.

Mail79a.

Maille, M., ''Methodes d'evaluation de la precision d'une mesure ou d'un calcul numerique.,'' *Rapport L.I.T.P.*, Universite P. et M. Curie, Paris, France, 1979.

Mair87a.

Mairson, H. and J. Stolfi, ''Reporting and Counting Intersections Between Two Sets of Line Segments,'' *Proceedings of NATO ASI on TFCG & CAD*, July 1987.

Mara73a.

Marasa, J. and D. Matula, ''A Simulative Study of Correlated Error Propagation in Various Finite-Precision Arithmetics,'' *IEEE Transactions on Computers*, vol. c-22, no. 6, pp. 587-597, June 1973.

Mile86a.

Milenkovic, V., ''Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic,'' *Intl. Workshop on Geometric Reasoning*, Oxford, England, 1986.

Mill80a.

Miller, Webb and Celia Wrathall, *Software For Roundoff Analysis of Matrix Algorithms,* Academic Press, NYC, 1980.

Moor66a.

Moore, R. E., *Interval Analysis,* Prentice Hall, Engelwood Cliffs, NJ, 1966.

Mudu86a.

Mudur, S., "Mathematical Elements for Computer Graphics," in *Advances in Computer Graphics I*, ed. F. Lillehagen, Springer-Verlag, 1986.

Ottm87a.

Ottmann, T., G. Thiemt, and C. Ullrich, "Numerical Stability of Geometric Algorithms," *Proceedings of the ACM Symposium on Computational Geometry*, pp. 119-125, June 1987.

Pres86a.

Press, W., B. Flannery, A. Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing,* Cambridge University Press, 1986.

Prog84a.

Program, Graphics, *g - an Object Space Hidden Surface Elimination Program,* Princeton University, Dept. of Computer Science , 1984.

Rams82a.

Ramshaw, Lyle, "The Braiding of Floating Point Lines," *CSL Notebook Entry, Xerox PARC*, October 1982.

Roge85a.

Rogers, David, *Procedural Elements for Computer Graphics,* McGraw-Hill Book Company, 1985.

Schwa.

Schwarz, J., "Guide to the C++ Real Library (Infinite Precision Floating Point)," *Unpublished Manuscript,* AT&T Bell Laboratories.

Sega85a.

Segal, M. and C. Sequin, "Consistent Calculations for Solids Modeling," *Proc. of the ACM Symposium on Computational Geometry*, pp. 29-38, 1985.

Sega85b.

Segal, M. and C. Sequin, "Consistent Calculations for Solids Modeling," *Proc. of the ACM Symposium on Computational Geometry*, pp. 29-38, 1985.

Sega88a.

Segal, M. and C. Sequin, "Partitioning Polyhedral Objects into Nonintersecting Parts," *IEEE Computer Graphics & Applications*, January 1988.

Sugi87a.

Sugihara, K., ''An Approach to Error-Free Solid Modeling,'' IMA Summer Program on Robotics, Inst. Math. and Applic., Univ of Minnesota, 1987.

Trus88a.

Truslove, K., ''The Implications of Tolerancing for Computer-Aided Mechanical Design,'' *Computer-Aided Engineering Journal*, pp. 79-85, April 1988.

Vand78a.

Vandergraft, J., *Introduction to Numerical Computations,* Academic Press, New York, 1978.

Vign78a.

Vignes, J., ''New Methods For Evaluating The Validity Of The Results Of Mathematical Computations,'' *Mathematics and Computers in Simulation XX*, pp. 227-249, 1978.

Vign84a.

Vignes, J., ''An Efficient Implementation of Optimization Algorithms,'' *Mathematics and Computers in Simulation XXVI*, pp. 243-256, 1984.

Vign87a.

Vignes, J., ''Zero Mathematique et Zero Informatique,'' *La Vie des Sciences*, vol. 4, no. 1, pp. 1-13, 1987.

Vign74a.

Vignes, J. and M. La Porte, ''Error Analysis In Computing,'' *Proceedings IFIP Congress*, pp. 610-614, Stockholm, 1974.

Wilk63a.

Wilkinson, J., *Rounding Errors in Algebraic Processes,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1963.

Yap88a.

Yap, C., ''A Geometric Consistency Theorem for a Symbolic Perturbation Scheme,'' *Proceedings of the ACM Symposium on Computational Geometry*, June 1988.