APPLIED COMPUTATIONAL GEOMETRY: TOWARDS ROBUST SOLUTIONS OF BASIC PROBLEMS

David Dobkin
Deborah Silver

CS-TR-192-88

December 1988

Applied Computational Geometry:

Towards Robust Solutions of Basic Problems [†]

David Dobkin

Department of Computer Science
Princeton University
Princeton, NJ 08544

Deborah Silver

Department of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08904

Abstract: *Geometric computations, like all numerical procedures, are extremely prone to roundoff error. However, virtually none of the numerical analysis literature directly applies to geometric calculations. Even for line intersection, the most basic geometric operation, there is no robust and efficient algorithm. Compounding the difficulties, many geometric algorithms perform iterations of calculations reusing previously computed data. In this paper, we explore some of the main issues in geometric computations and the methods that have been proposed to handle roundoff errors. In particular, we focus on one method and apply it to a general iterative intersection problem. Our initial results seem promising and will hopefully lead to robust solutions for more complex problems of applied computational geometry.*

1. Introduction & Motivation

As algorithmic techniques in computational geometry and graphics algorithms mature, attention is focussed on the problem of technology transfer. The goal is to determine which theoretically fast algorithms actually work well in practice and to find methods of turning the efficient into the practical. 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 exceptions. 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 depending upon the application. For graphical output, anywhere from 3-12 significant digits of precision may suffice, whereas for other areas, more may be needed. Having a program compute twenty digits of precision where only three are needed is

[†] This work was supported in part by a National Science Foundation grant CCR-87-00917 and by a Hewlett-Packard Faculty Development Grant to the second author.

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 target its operations to a user specified end 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, dumping core, or causing infinite looping.

However, this is not a trivial issue. Forrest argues that there are no robust algorithms for even the simple and basic problem of line segment intersection [Forr87a]. The recent flurry of activity on this problem confirms his belief. Knott and Jou [Knot87a] give methods 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. For example, in several hidden line-surface elimination algorithms, polygon intersection is performed by calculating the intersection of the *computed intersection of various polygons* with other polygons. These cascading calculations suffer from roundoff error as well as from computing with inexact data as the calculations progress (propagation 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, so that the calculated line is a shifted version of the ''real'' one causing any further computations with that line segment to be off no matter how exact the arithmetic functions are (of course, if all the calculations are precise this problem would not exist). As the calculations progress, 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]

This is similar to the following problem, which we consider in our paper:

> 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 elimination and many other computations in computer graphics (ray tracing, CAD-CAM). Indeed, our attention was focussed on this problem because of our frustration with *ad hoc* methods being used 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.

In this article, an initial attempt at approaching roundoff issues in cascading geometric computation is presented. The organization of this paper is as follows. The second section presents a brief review of some existing methods dealing with roundoff error in geometric computations, and a description of a sample geometric problem with one of the methods singled out for its applicability to this problem. The third, fourth, and fifth sections contain the application, analysis, and conclusion. The results of this work are four-fold. First, we have explored the various approaches to the issue of robustness in general and have demonstrated a method of computing precision in an ongoing geometric computation. We have also analyzed the cost of backtracking and means of avoiding it. Third, we have proposed an empirical solution for a cascaded line intersection procedure. And lastly, we have presented insights into the problem which will hopefully spur additional research and applications.

2. Literature Survey

The most widely applied solution to the problem of roundoff error 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* values, and in most instances, will only solve a set of problems temporarily and does not attack the underlying cause of the roundoff error. Needless to say, this approach is far from robust and consistent.

Line intersection calculation can be viewed as being either of geometric or of numeric flavor, and the attempts at coping with the roundoff error problem have taken one of these two approaches. The geometric flavored solutions strive to maintain correct topological information using finite precision. This is accomplished with special functions and data structures to keep the geometric objects in a consistent state. To overcome floating point error, *epsilon procedures* are used to handle the ambiguous cases and to keep the objects "far enough" apart. Milenkovic [Mile86a] proposed two methods for *verifiable implementations of geometric algorithms using finite precision*. The first is *data normalization* which alters the objects by *vertex shifting* and *edge cracking* to maintain a distance of at least ε (determined by machine roundoff error) between the geometric structures. The second method is called the *hidden variable method*, which constructs configurations of objects that belong in an infinite precision domain, without actually representing these infinite precision objects, by modeling approximation to geometric lines with monotonic curves. Segal and Sequin's [Sega88a] method introduces a *minimum feature size* and *face thickness* to objects and then either merges or pulls apart those objects that lie within the minimum feature size of each other. Hoffmann, Hopcroft, and Karasick [Hoff88a] add symbolic reasoning to compensate for numerical uncertainties when performing set operations on polyhedral solids. Related to line intersection, Ramshaw [Rams82a] shows how floating point line segments can appear to "braid" by intersecting each other more than once. To correct this, Greene and Yao [Gree86a] transform geometric objects from the continuous domain to the discrete domain and perform all the calculations in the discrete domain. Line segments are treated as a set of raster points (these are the points used by line drawing algorithms) and the line is the shortest path within this *envelope* of points. The line-path is controlled with hooks which serve to direct the line to pass through specified grid points in order to insure that it will intersect certain lines while not crossing others.

The numeric flavored solutions consider line intersection primarily as a set of numerical calculations, as opposed to operations on geometric objects, and borrow from classical numerical analysis, i.e. roundoff-error analysis. Pioneered by Wilkinson [Wilk63a], this approach involves forward and backward error analysis and determination of condition numbers for a particular set of calculations. The condition numbers can alert the programmer or user to possible bad sets of data or 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.). Although this type of analysis is basic to a first approach at combatting roundoff error, it is not helpful in dealing with the buildup of unavoidable roundoff error. Unfortunately, we know of no numerical analysis literature regarding the accumulation of roundoff error in cascading processes such as we consider.

Many tackle the difficulty of roundoff error by proposing modified floating point systems. Kulisch and Miranker introduce a dot product function that performs the dot product of two vectors rounding only at the end instead of after each individual multiplication and addition [Kuli81a]. Ottmann, Thiemt, and Ullrich [Ottm87a] show how to implement "stable" geometric primitives with this dot product function. A popular approach is interval arithmetic [Moor66a] which treats a rounded real number as an interval between its two bounding representable real numbers, and calculations are performed on this interval widening the resulting interval as necessary. 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. In their work, 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. Knott and Jou [Knot87a] also use interval arithmetic to *determine correctly whether two line segments intersect* and, if that fails, resort to multiple-precision floating-point arithmetic. Another method is that of Vignes and La Porte [Vign74a] which takes a stochastic approach to evaluating the number of significant digits in a computed result. Their method generates a subset of all the possible computable results of a function and uses that subset to determine properties of the entire set. Attempting to avoid floating point computations altogether (which seems to be widely recommended advice), additional proposals include systems based upon rational arithmetic and symbolic computation. In addition, different floating point systems (implementation in hardware) vary in their performance with regards to roundoff error, and there has been work documenting those differences [Kaha88a, 85a]. (For a more comprehensive survey see [Hoff88b, Silv88a]).

Although useful in various situations, the aforementioned methods are of necessity flawed when applied to cascading intersection calculations. The geometric solutions are hard to implement and not directly applicable to this problem. 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. Interval arithmetic gives overly pessimistic results since the intervals grow much larger as the computations progress. While the geometric objects being operated on in cascading intersections generally become smaller, the intervals become bigger, causing in many instances, the intervals to be larger than the objects being manipulated (techniques for narrowing the computed intervals must be used to obtain satisfactory results).

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

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. The *in* iteration computes the intersection of the pentagon's diagonals resulting in a smaller inverted pentagon. The 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 Figure 1). In each iteration following the first, the data used are those calculated by the previous iteration. An iteration *in* and then *out* is an identity function; therefore, after an equal amount of *ins* and *outs* the differences between the 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*. Sometimes it is impossible to maintain any precision in the calculated data.

The key to the difficulty in the *pentagon problem*, as well as some other 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 cascading calculations without being overly pessimistic. However, one method that does enable easy and accurate prediction of error generated during compounded calculations is the Permutation-Perturbation method of Vignes

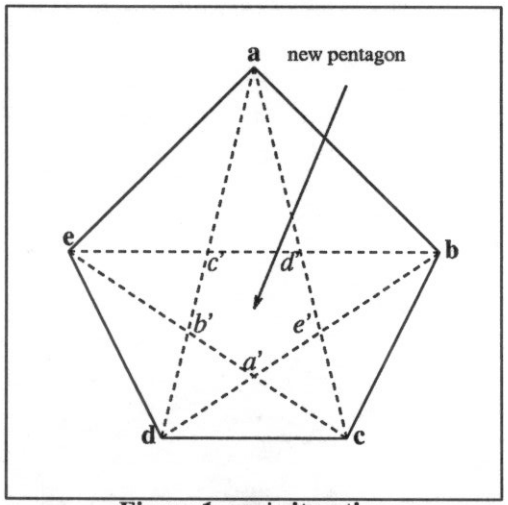

Figure 1: an *in* iteration

and La Porte [Vign74a]. In what follows, we describe their method in detail, and discuss its application to the problem of cascading intersections with emphasis on efficiency, accuracy, and their tradeoffs.

2.2. The Permutation-Perturbation Method

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 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.* [1] Let P_{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 $2^k \times P_{op}$.

The number of significant digits of a computed value can be determined by [2]

$$10^{-C} = \frac{|x - x_c|}{|x|}$$

where C is the number of significant digits, x_c is the computed value, and x is the "exact" result. This quantity is generally expressed with the absolute error $(x - x_c)$ divided by x_c instead of x, which is valid when x_c is a reasonable approximation to x (see below).

Vignes and La Porte use this formula to determine the precision of computed value. However, they attempt to estimate the error since the exact error may not be known at computation time. The formula used is $\frac{\delta}{\overline{R}}$ where $\overline{R}$ is the mean of the population **R** and δ the standard deviation. Both δ and $\overline{R}$ can be evaluated probabilistically by drawing samples from **R**. (For more complete detail see [Vign78a, Faye85a, Vign74a].)

[1] for example, the four numbers

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

when added left to right, using four digit arithmetic, results in the exact sum $.6755\times10^1$. However, adding from right to left produces $.7000\times10^1$. (This example also illustrates subtractive cancellation.) [Vand78a]

[2] This is the formula used for computing the relative error.

This analysis rests upon two hypotheses: 1) that $\bar{R}=r$ (r is the exact result) to within an error smaller than or at worst comparable with δ (see [Mail79a]), and 2) that **R** is better approximated by a continuous distribution than by a small discrete population. The first implication can be false on those rare occasions when the computation in question comes as close to a singularity as it can without actual collision. The second may cause problems when only a few among the many rounding errors contribute the bulk of the error in the final result (undersampling **R**) [Kaha88b]. However, for the majority of cases, this method is likely to give a fair indication of a computation's typical accuracy.

3. Application

The application of the Permutation-Perturbation method to the *pentagon problem* was straightforward. On each iteration, all intersection points were calculated three to four times using different permutations/perturbations of the intersection code (care must be taken when performing permutations), and the number of significant digits of the average was computed with the formula of La Porte and Vignes (see above). This was done for all five pentagon vertices (ten 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 Figure 2).

(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.* 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... [3]

Normally, the best computed result of an iteration was the average of the three (four) 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 different permutation used one of the results). This is equivalent to performing all the iterations as one computation but stopping it along the way for precision determination. Thereby enabling 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.

After all the iterations were completed, the error incurred during the computations was calculated (since the original pentagon was given). The exact error was then compared with the computed predicted error to analyze the performance of the Permutation-Perturbation method. Fortunately, the Permutation-Perturbation method proved to be an accurate predictor of roundoff error buildup in the calculated results and was within one digit of the actual error (see Figure 2).

4. 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 can 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 (as expected). The initial decline of significant digits (or increase in error) began by slowly curving downwards 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 causing any error in the input data to be 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.

[3] It is not always possible to extend outwards starting with the initial pentagon e.g. if two sides are parallel to each other. However, sometimes it may be possible 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.

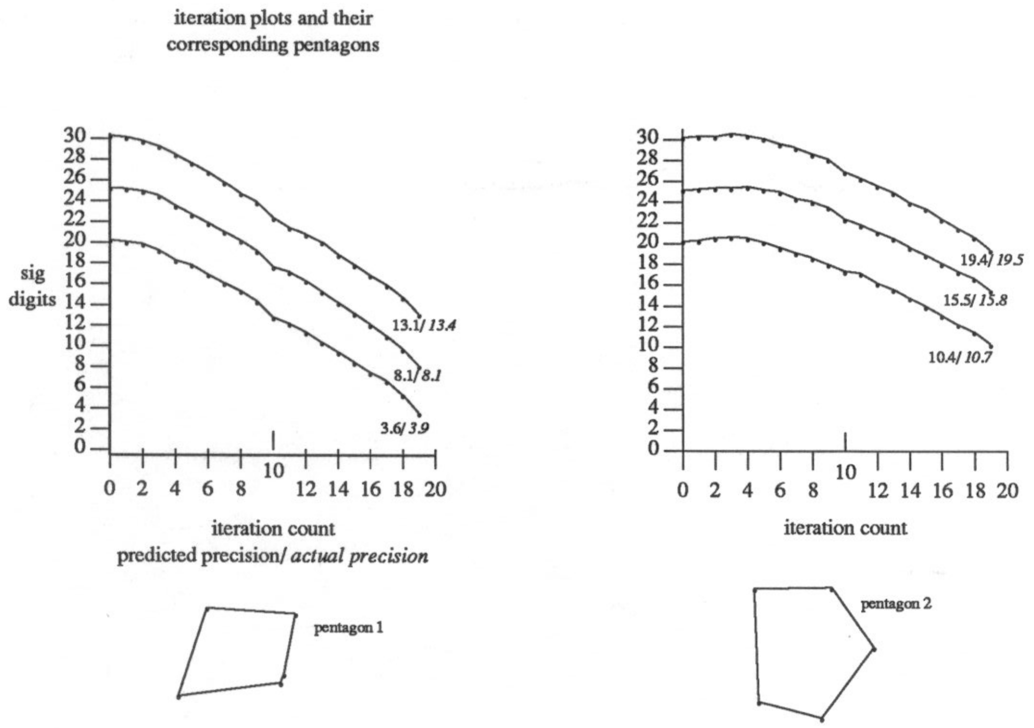

Figure 2: in-10 out-10 series

Furthermore, the iterations are rotation invariant, namely, a pentagon and its rotated version resulted in similar plots. For the same reason, 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. [4] Other series combinations (in-5 out-5 etc..) performed in the same manner as the in-10 out-10 series; the curves 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.

Based upon the experimentation, the Permutation-Perturbation method proved to be helpful in predicting the amount of error accumulated during cascading line intersection calculations. Although it has an associated cost of a factor of three or four times that of doing nothing, 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) which is no small achievement when dealing with numerical algorithms. It requires no special mathematical functions for the basic arithmetic operations and no special 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

---

[4] intersecting perpendicular lines gives a more accurate result than intersecting those that are close to parallel [Forr85a].

error estimator but also to set the "fuzz" values in a program. Finally, this method provides an accurate estimate of the errors accumulated during the computations without being overly pessimistic or optimistic.

4.1. 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. [5]

4.2. 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 questions 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 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:

4.3. 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

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

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.

4.4. 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.

4.5. 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 needed 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.

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).

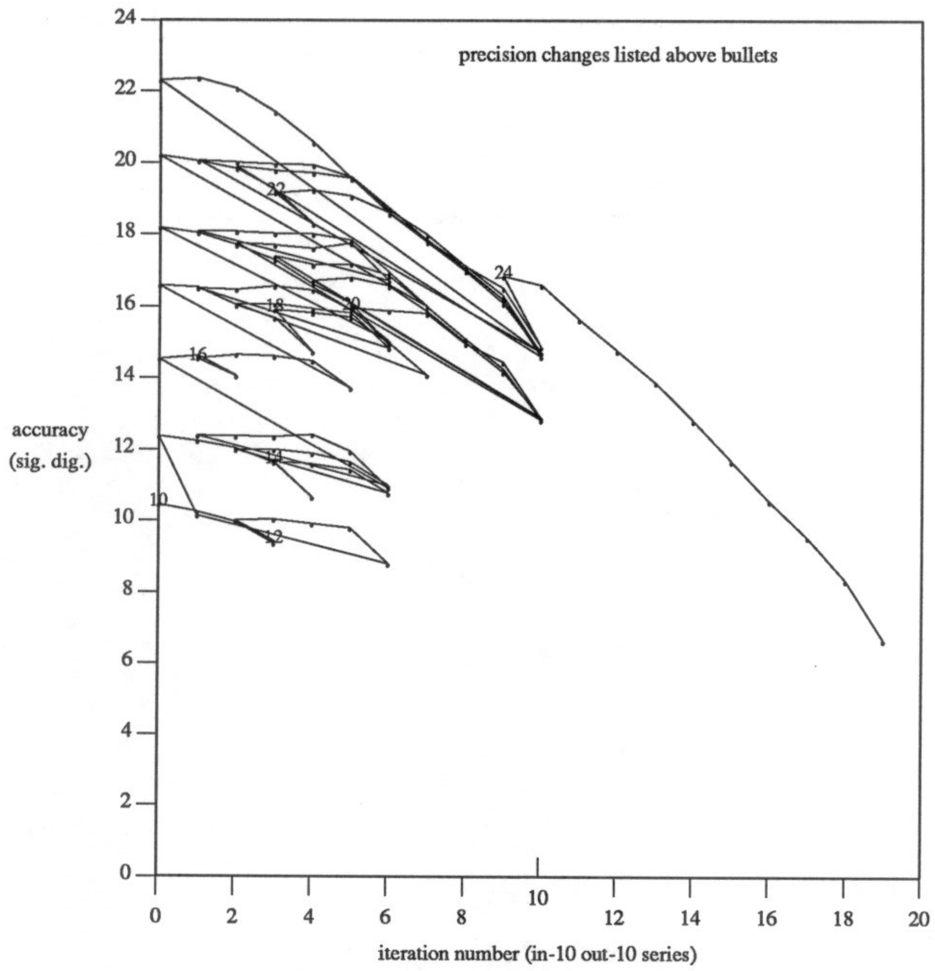

Figure 1: ***zigzagging* example**

4.6. Problem 4

It is important to note that the method described here accomplishes two things. First it produces the ''best'' answer, and second, it estimates the accuracy of that answer. There may be methods which perform the first function more efficiently (see [85a]), however, they do not perform the second function as well.

Furthermore, the Vignes' method is stochastic and is therefore prone to potential pitfalls which are inherent in any such approach. There is a tradeoff in numerical analytic techniques between reliability and accuracy. While condition numbers and interval arithmetic are reliable, they are worst case approximations and in many instances cannot be easily applied. This technique is accurate, although there may be unusual cases that slip through. However, it is very likely to give a fair indication of a computation's typical accuracy under normal circumstances [Kaha88b]. A further advantage is that it is less cumbersome to use than the other approaches. The possibility to do better with some other method is as yet an unexplored option.

4.7. Problem 5

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.

We have begun to test this method with a hidden surface elimination routine (based on polygon intersections) and have found that the precision degrades slower than for the *pentagon problem.* The reason for this is that generally when a ''new'' line is formed only one of the vertices is ''newly'' computed, while the other is from the original line.

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.

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 calculations 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. Conclusions and 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 solution are proposed. Our 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. Using an error estimator coupled 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, our solution seems to be less expensive than many others.

6. Acknowledgements

We would like to thank Prof. William Kahan for his helpful comments and suggestion.

desired end precision (for all graphs) = 5

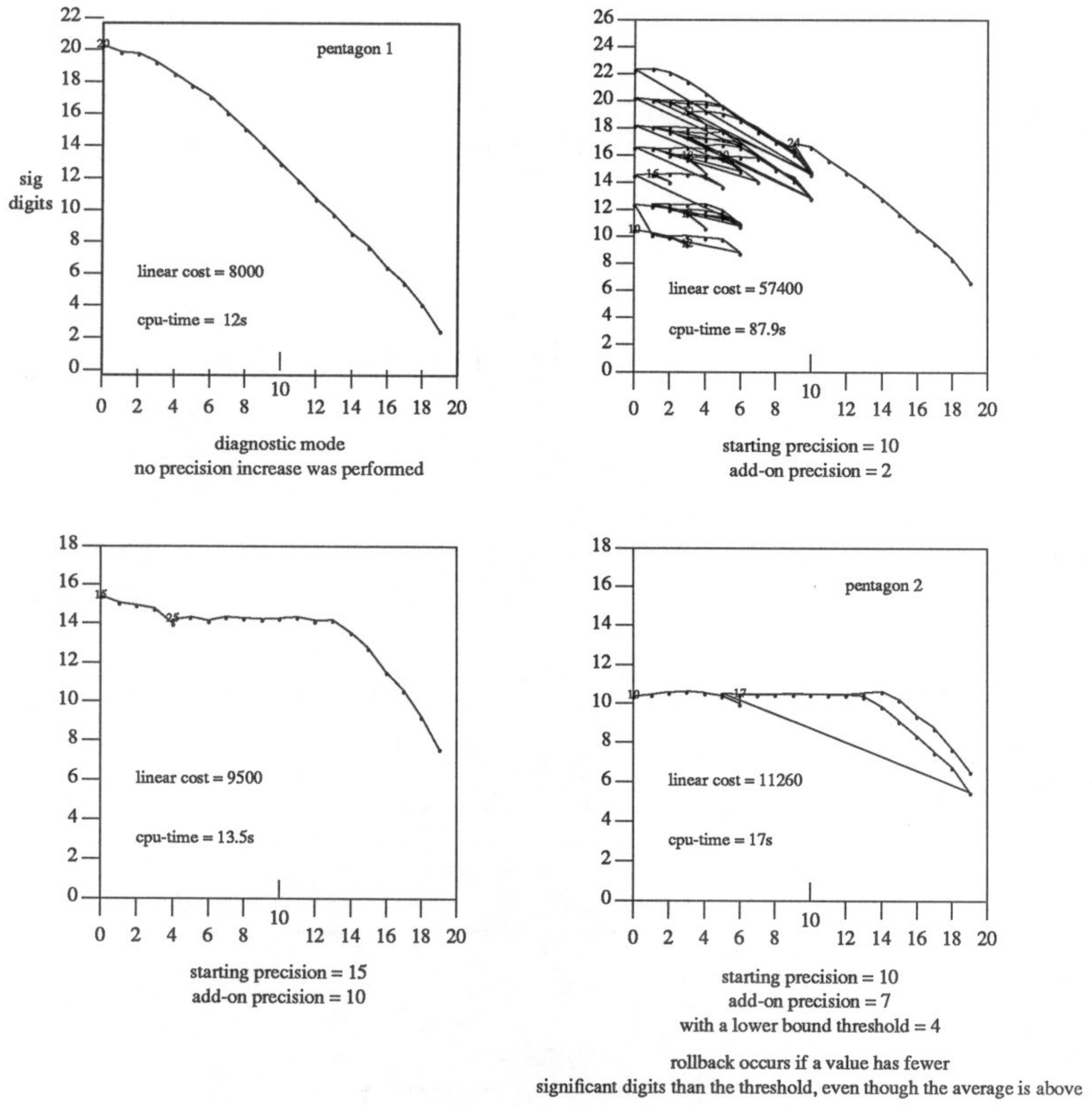

Figure 4: Cost Function Examples

References

85a. ''IEEE Standard for Binary Floating-Point Arithmetic,'' ANSI/IEEE Std 754-1985, 1985.

Demm86a.
Demmel, J., ''On Error Analysis in Arithmetic With Varying Relative Precision,'' TR 251, NYU Dept. of Computer Science, October 1986.

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.

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.

Kaha88a.
Kahan, W., ''A Computer Program with Almost No Significance,'' *Work in progress*, 1988.

Kaha88b.
Kahan, W., *Private Communication*, October 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.

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.

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.

Rams82a.
Ramshaw, Lyle, ''The Braiding of Floating Point Lines,'' *CSL Notebook Entry, Xerox PARC*, October 1982.

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.

Sega88a.
Segal, M. and C. Sequin, ''Partitioning Polyhedral Objects into Nonintersecting Parts,'' *IEEE Computer Graphics & Applications*, January 1988.

Silv88a.
Silver, D., ''Geometry, Graphics, & Numerical Analysis,'' *Ph.D. Thesis (in preparation)* , Princeton University, 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.

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.