

This paper will appear at the Symposium on Computational Geometry, June 6-8, 1988.

RECIPES FOR GEOMETRY & NUMERICAL ANALYSIS

PART I: AN EMPIRICAL STUDY

David Dobkin  
Deborah Silver

CS-TR-144-88

March 1988

# Recipes for Geometry & Numerical Analysis -

## Part I: An Empirical Study <sup>†</sup>

*David Dobkin  
Deborah Silver*

Department of Computer Science  
Princeton University  
Princeton, NJ 08544

### 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 computational geometry.*

March 22, 1988

---

<sup>†</sup> 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.

# Recipes for Geometry & Numerical Analysis -

## Part I: An Empirical Study <sup>†</sup>

David Dobkin  
Deborah Silver

Department of Computer Science  
Princeton University  
Princeton, NJ 08544

### 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 not 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 on the application. For graphical output, three or four significant digits of precision may suffice, whereas for other areas, much more may be needed. Having a program compute twenty digits of precision where 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 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

<sup>†</sup> 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.

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. 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 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.

In this article, an initial attempt at approaching roundoff issues in cascading geometric computation is presented. A companion paper (Part II) will provide a more theoretical exploration of this issue. The organization of this paper is as follows. The second section contains a short list of some existing methods that deal with roundoff error 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 section contain the application, analysis, and conclusion. The results of this work are fourfold. First, we have demonstrated a method of computing precision in an ongoing geometric computation. We have also explored 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 hopefully presented insights into the problem which the reader can apply elsewhere.

## 2. Finding the Best Ingredients

### Previous attempts

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 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. However, these fixes are based upon theoretical principles (although not necessarily applied with any). Those principles relating to line intersection are listed below.

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 ("attempted 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  $\epsilon$  (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 [Sega85a] 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. Hoffman, Hopcroft, and Karasick [Hoff87a] 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 intersect each other more than once. To avoid that, Greene and Yao's [Gree86a] approach transforms the geometric objects from the continuous domain to the discrete domain and performs 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 to determine the 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] applied interval arithmetic to the processing of geometric objects. In their work, they define several geometric functions with interval computations and devise new algorithms 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 [Korn83a] and symbolic computation.

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 (as opposed to computing the condition number of a particular set of calculations, ignoring the unknown and substantial amount of error which has accumulated in the data before these calculations). 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.

### The First Ingredient: 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. 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 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.

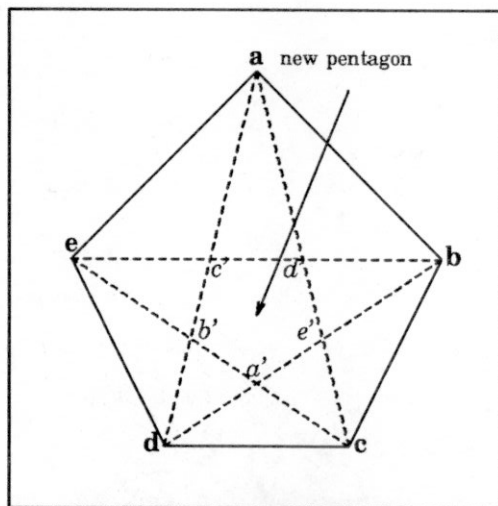


Figure 1: an *in* iteration

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. 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]. 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.

### The Second Ingredient: 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*.<sup>[1]</sup> 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

$$\epsilon_r = R_0 - r.$$

[1] for example, the four numbers

$$.1025 \times 10^4, \quad -.9112 \times 10^3, \quad -.9773 \times 10^2, \quad -.9315 \times 10^1$$

when added left to right, using four digit arithmetic, results in the exact sum  $.6755 \times 10^1$ . However, adding from right to left produces  $.7000 \times 10^1$ . (This example also illustrates subtractive cancellation.) [Vand78a]

If  $\bar{\epsilon}_r$  is the mean estimate of  $\epsilon_r$ ,  $\bar{R}$  the mean value of all the elements of R, and  $\delta$  the standard deviation of these elements, then

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

Since r can be any element in the population R, then <sup>[2]</sup>

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

Therefore,

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

and so

$$\frac{\bar{\epsilon}_r}{|R_0|} = 10^{-C}$$

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 enough [Mail79a]. (For more complete detail see [Vign78a, Faye85a, Vign74a]. )

### 3. Mixing the First Two Ingredients

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 of the intersection code, 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... <sup>[3]</sup>

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. 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 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.

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

[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.



After all the iterations were completed, the final computed vertices were tested against the original vertices 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 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).

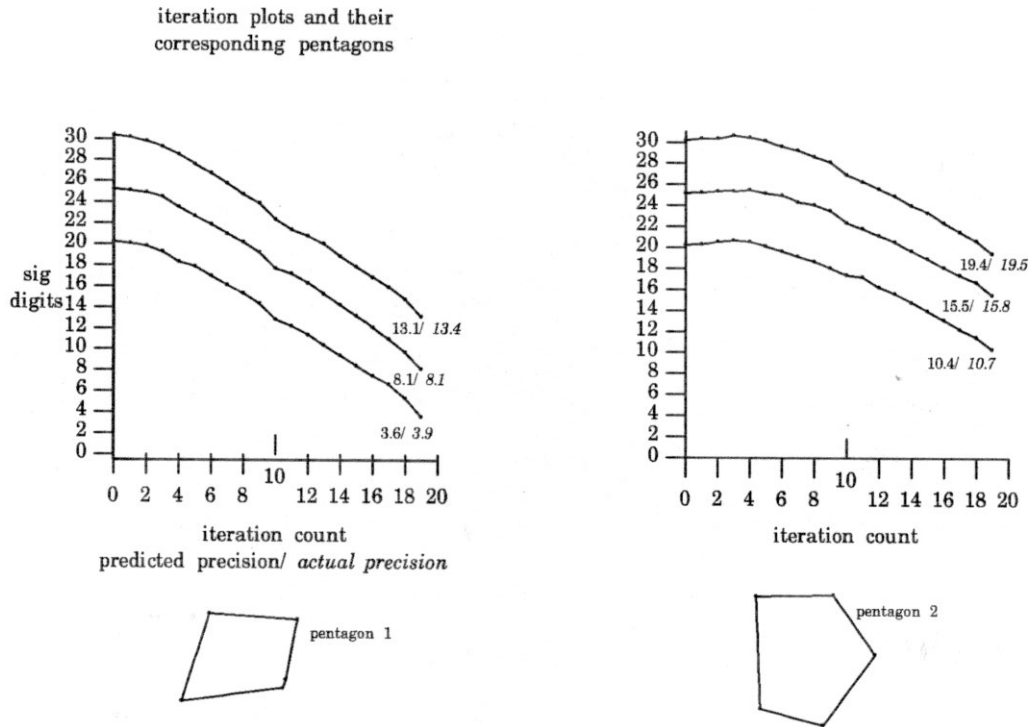


Figure 2: in-10 out-10 series

#### 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. 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 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. <sup>[4]</sup> Other series combinations (in-5 out-5 etc..) performed in the same manner as 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.

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) 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 special 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 the errors accumulated during the computations without being overly pessimistic or optimistic.

### The Third and Final Ingredient

To increase the number of significant digits in a computed value, one must increase the precision used to calculate that value. This can be accomplished with any multiple precision package.

### 4. Preparing the stew

Now that all the ingredients are assembled, namely, the sample problem, the error estimator, and the multiple precision package, they must be blended into an "edible" format. Although we have the ingredients for the stew, we do not have an exact recipe. Namely, these 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.

---

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

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. 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:

**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. For this problem, since the first two quantities are known (or computable) and all the iterations affect the precision in a similar manner, we can estimate the decay in precision with a linear rule. That is, let  $a$  digits remain after the computation,  $b$  digits exist before with  $i$  iterations left to be completed. This being the case at some point in the iteration process, we estimate the number of significant digits in the final result as

$$a - i^*(b - a).$$

The value in parenthesis represents the decline in significant digits of the last iteration. Alternately, a larger "history" can be used to produce a more accurate 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.

**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. 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 *zigzagging* or *thrashing*, i.e. repeated backtracking with increased precision until the correct amount is finally attained (see Figure 3).

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.

**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.

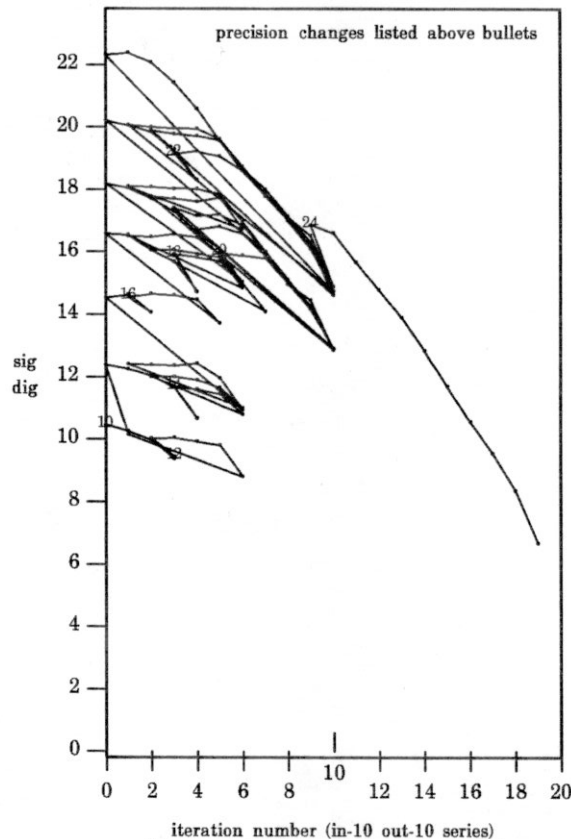


Figure 3: zigzagging example

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 effects 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 Figure 4). 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.

**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).

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. However, a precision estimator is very useful in determining the values for the various *epsilons* which appear 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. Furthermore, this method is very useful as a debugging tool to track and identify numerical problems which cause program failures. (See [Silv88a] for more detail.)

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*. Further study is necessary to fully resolve these questions.

## 5. Conclusions and Leftovers

These are just some of the major issues that arose during the initial tests (tastes) of the "stew". 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. 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. 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, our solution seems to be less expensive than many others.

desired end precision (for all graphs) = 5

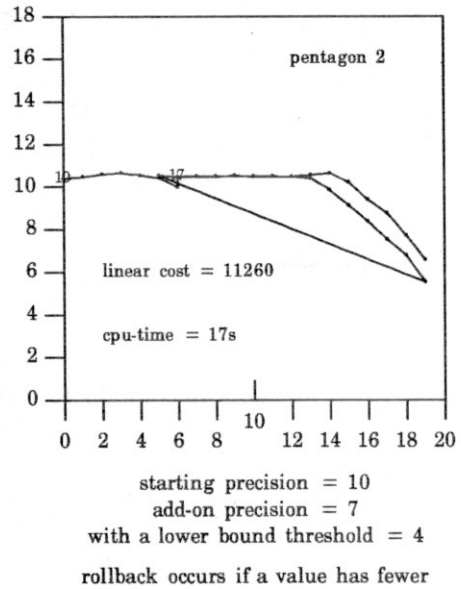
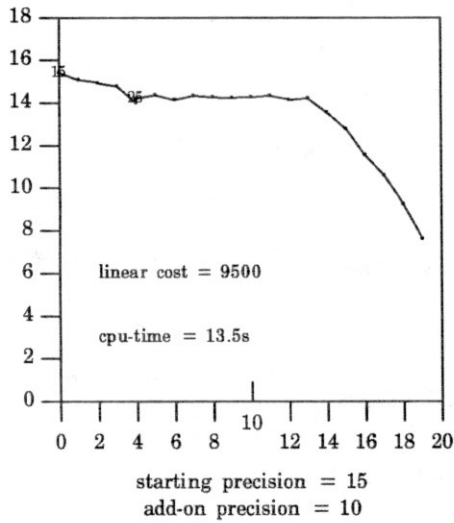
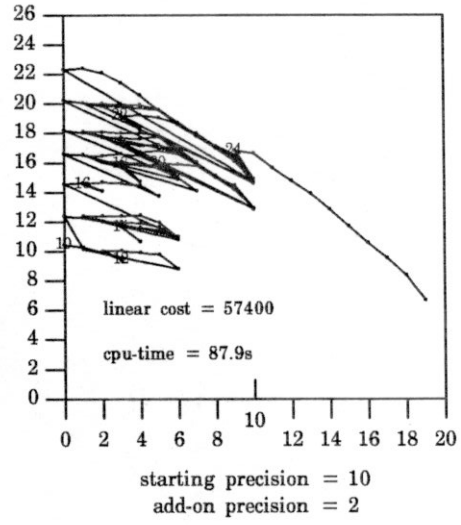
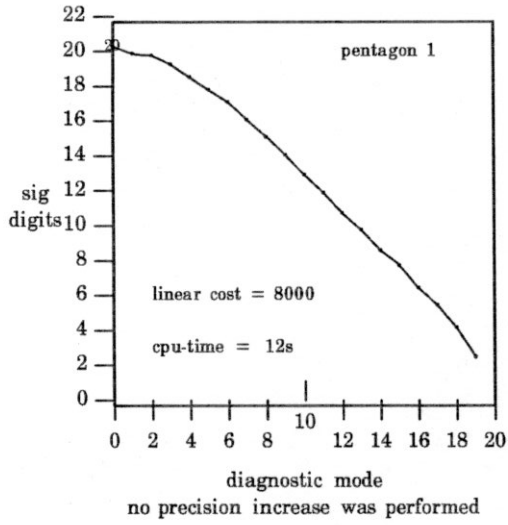


Figure 4: Cost Function Examples

## References

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. II Cioceo, Italy, July 1987.

Gree86a.

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

Hoff87a.

Hoffmann, C., J. Hopcroft, and M. Karasick, "Robust Set Operation on Polyhedral Solids," *Unpublished Manuscript*, 1987.

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.

Korn83a.

Kornerup, P. and D. Matula, "Finite Precision Rational Arithmetic: An Arithmetic Unit," *IEEE Transaction on Computers*, vol. c-32, no. 4, April 1983.

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," *Unpublished Manuscript*, 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 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.

Sega85a.

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

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.