Robust Adaptive Floating-Point Geometric Predicates

Jonathan Richard Shewchuk
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
jrs@cs.cmu.edu

Abstract

Fast C implementations of four geometric predicates, the 2D and 3D orientation and incircle tests, are publicly available. Their inputs are ordinary single or double precision floating-point numbers. They owe their speed to two features. First, they employ new fast algorithms for arbitrary precision arithmetic that have a strong advantage over other software techniques in computations that manipulate values of extended but small precision. Second, they are adaptive; their running time depends on the degree of uncertainty of the result, and is usually small. These algorithms work on computers whose floating-point arithmetic uses radix two and exact rounding, including machines that comply with the IEEE 754 floating-point standard. Timings of the predicates, in isolation and embedded in 2D and 3D Delaunay triangulation programs, verify their effectiveness.

1 Introduction

Algorithms that make decisions based on geometric tests, such as determining which side of a line a point falls on, often fail when the tests return false answers because of roundoff error. The easiest solution to many of these robustness problems is to use software implementations of exact arithmetic, albeit often at great expense. The goal of improving the speed of correct geometric calculations has received much recent attention, but the most promising proposals take integer or rational inputs, typically of small precision. These methods do not appear to be usable if it is convenient or necessary to use ordinary floating-point inputs.

This paper describes two techniques for implementing fast exact (or extended precision) geometric calculations, and demonstrates them with implementations of four commonly used geometric predicates, the 2D and 3D orientation and incircle tests, available on the Web at "http://www.cs.cmu.edu/∼quake/robust.html". The orientation test determines whether a point lies to the left of, to the right of, or on a line or plane; it is an important predicate used in many (perhaps most) geometric algorithms. The incircle test determines whether a point lies inside, outside, or on a circle or sphere, and is used for Delaunay triangulation [8]. Inexact versions of these tests are vulnerable to roundoff error, and the wrong answers they produce can cause geometric algorithms to hang, crash, or produce incorrect output. Although exact arithmetic banishes these difficulties, it is common to hear reports of implementations being slowed by factors of ten or more as a consequence. For these reasons, computational geometry is an important arena for evaluating extended precision arithmetic schemes.

The first technique is a set of algorithms, several of them new, for performing arbitrary precision arithmetic. They differ from traditional methods in two ways. First, they gain speed by relaxing the usual requirement that extended precision numbers be normalized to fixed digit positions. Second, there is no cost to convert floating-point numbers to a specialized extended precision format. The method has its greatest advantage in computations that process values of extended but small precision (several hundred or thousand bits), and seems ideal for computational geometry. The method was largely developed by Priest [12, 13], who designed similar algorithms that run on a wide variety of floating-point architectures, with different radices and rounding behavior. I have made significant speed improvements by relaxing Priest's normalization requirement and optimizing for radix two with exact rounding. This specialization is justified by the wide acceptance of the IEEE 754 floating-point standard.

The second technique is to exploit features of the predicates that frequently make it possible for them to return correct answers without completing an exact computation. The orientation and incircle tests evaluate the sign of a matrix determinant. It is significant that only the sign, and not the magnitude, of the determinant is needed. Fortune and Van Wyk [5] take advantage of this fact by using a floating-point filter: the determinant is first evaluated approximately, and only if forward error analysis indicates that the sign of the approximate result cannot be trusted does one use an exact test. I carry their suggestion to its logical extreme by computing a sequence of successively more accurate approximations to the determinant, stopping only when the accuracy of the sign is assured. To reduce computation time, each approximation reuses a previous, less accurate computation if it is economical to do so. This adaptive approach is described in Section 4, and its application to the orientation and incircle

Supported in part by the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship and by the National Science Foundation under Grant ASC-9318163.

tests is described in Section 5.

2 Related Work

There are several exact arithmetic schemes designed specifically for computational geometry; most are methods of exactly evaluating the sign of a determinant. Clarkson [3] proposes an algorithm for using floating-point arithmetic to evaluate the sign of the determinant of a small matrix of integers. A variant of the modified Gram-Schmidt procedure is used to improve the conditioning of the matrix, so that the determinant can subsequently be evaluated safely by Gaussian elimination. The 53 bits of significand available in IEEE double precision numbers are sufficient to operate on 10×10 matrices of 32-bit integers. Clarkson's algorithm is naturally adaptive; its running time is small for matrices whose determinants are not near zero[1].

Recently, Avnaim, Boissonnat, Devillers, Preparata, and Yvinec [1] proposed an algorithm to evaluate signs of determinants of 2×2 and 3×3 matrices of p-bit integers using only p and $(p + 1)$-bit arithmetic, respectively. Surprisingly, this is sufficient even to implement the insphere test (which is normally written as a 4×4 or 5×5 determinant), but with a handicap in bit complexity; 53-bit double precision arithmetic is sufficient to correctly perform the insphere test on points having 24-bit integer coordinates.

Fortune and Van Wyk [5, 6] propose a more general approach (not specific to determinants, or even to predicates) that represents integers using a standard exact arithmetic technique with digits of radix 2^{23} stored as double precision floating-point values. (53-bit double precision significands make it possible to add several products of 23-bit integers before it becomes necessary to normalize.) Rather than use a general-purpose arbitrary precision library, they have developed LN, an expression compiler that writes code to evaluate a specific expression exactly. The size of the operands is arbitrary, but is fixed when LN generates a C++ implementation of the expression. An expression can thus be used to generate several functions, each for arguments of different bit lengths. Because the expression and the bit lengths of all operands are known in advance, LN can tune the exact arithmetic aggressively, eliminating loops, function calls, and memory management. The running time of the function thus produced depends on the bit complexity of the inputs. Fortune and Van Wyk report an order-of-magnitude speed improvement over the use of libraries (for equal bit complexity). Furthermore, the expression compiler garners another speed improvement by installing floating-point filters wherever appropriate, calculating static error bounds automatically.

Karasick, Lieber, and Nackman [9] report their experiences optimizing a method for determinant evaluation using rational inputs. Their approach reduces the bit complexity of the inputs by performing arithmetic on intervals (with low precision bounds) rather than exact values. The determinant thus evaluated is also an interval; if it contains zero, the precision is increased and the determinant reevaluated. The procedure is repeated until the interval does not contain zero (or contains only zero), and the sign of the result is certain. Their approach is thus adaptive, although it does not appear to use the results of one iteration to speed the next.

Because the Clarkson and Avnaim et al. algorithms are effectively restricted to low precision integer coordinates, I do not compare their performance with that of my algorithms, though theirs may be faster. Floating-point inputs are more difficult to work with than integer inputs, partly because of the potential for the bit complexity of intermediate values to grow more quickly. (The Karasick et al. algorithm also suffers this difficulty, and is probably not competitive against the other techniques discussed here, although it may be the best existing alternative for algorithms that require rational numbers, such as those computing exact line intersections.) When it is necessary for an algorithm to use floating-point coordinates, the aforementioned methods are not currently an option (although it might be possible to adapt them using the techniques of Section 3). I am not aware of any prior literature on exact determinant evaluation that considers floating-point operands.

3 Arbitrary Precision Floating-Point

3.1 Background

Most modern processors support floating-point numbers of the form $\pm$significand $\times$ 2^{exponent}. The significand is represented by a p-bit binary number of the form $b.bbb\ldots$ (where each b denotes a single bit), plus one additional bit for the sign. This paper does not address issues of overflow and underflow, so I allow the exponent to be an integer in the range $[-\infty, \infty]$. (Fortunately, many applications have inputs that fall within a circumscribed exponent range and will not overflow or underflow.) See the survey by Goldberg [7] for a detailed explanation of floating-point storage formats, particularly the IEEE 754 standard.

Most arbitrary precision libraries store numbers in a *multiple-digit* format, consisting of a sequence of digits (usually of large radix, like 2^{32}) coupled with a single exponent. A freely available example of the multiple-digit approach is Bailey's MPFUN package [2], a sophisticated portable multiprecision library that uses digits of machine-dependent radix (usually 2^{24}) stored as single precision floating-point values. An alternative is the *multiple-term* format, wherein a number is expressed as a sum of ordinary floating-point words, each with its own significand and exponent [12]. This approach has the advantage that the result of an addition like $2^{300} + 2^{-300}$ (which may well arise in a determinant computation with machine precision floating-point inputs) can be stored in two words of memory, whereas the multiple-digit approach will use at least 601 bits to store the sum, and incur a corresponding speed penalty when performing arithmetic with it.

For the algorithms herein, each arbitrary precision value is

[1] The method presented in Clarkson's paper does not work correctly if the determinant is exactly zero, but Clarkson (personal communication) notes that it is easily fixed. "By keeping track of the scaling done by the algorithm, an upper bound can be maintained for the magnitude of the determinant of the matrix. When that upper bound drops below one, the determinant must be zero, since the matrix entries are integers, and the algorithm can stop."

expressed as an *expansion*[2] $x = x_n + \cdots + x_2 + x_1$, where each x_i is called a *component* of x and is represented by a floating-point value with a p-bit significand. To impose some structure on expansions, they are required to be *nonoverlapping* and ordered by magnitude (x_n largest, x_1 smallest). Two floating-point values x and y are nonoverlapping if the least significant nonzero bit of x is more significant than the most significant nonzero bit of y, or vice-versa; for instance, the binary values 1100 and -10.1 are nonoverlapping, whereas 101 and 10 overlap[3]. The number zero does not overlap any number. An expansion is nonoverlapping if all its components are mutually nonoverlapping. Note that a number may be represented by many possible nonoverlapping expansions; consider $1100 + -10.1 = 1001 + 0.1 = 1000 + 1 + 0.1$. A nonoverlapping expansion is desirable because it is easy to determine its sign (take the sign of the largest component) or to produce a crude approximation of its value (take the largest component).

Multiple-term algorithms can be faster than multiple-digit algorithms because the latter require expensive normalization of results to fixed digit positions, whereas multiple-term algorithms can allow the boundaries between terms to wander freely. Boundaries are still enforced, but can fall at any bit position. In addition, it usually takes time to convert an ordinary floating-point number to the internal format of a multiple-digit library, whereas that number *is* an expansion of length one. Conversion overhead can be significant for small extended precision calculations.

The central conceptual difference between standard multiple-digit algorithms and the algorithms described herein is that the former perform exact arithmetic by keeping the bit complexity of operands small enough to avoid roundoff error, whereas the latter allow roundoff to occur, then account for it after the fact. To measure roundoff quickly and correctly, a certain standard of accuracy is required from the processor's floating-point units. The algorithms presented herein rely on the assumption that addition, subtraction, and multiplication are performed with *exact rounding*. This means that if the exact result can be stored in a p-bit significand, then the exact result is produced; if it cannot, then it is rounded to the nearest p-bit floating-point value (with ties broken arbitrarily). For instance, in four-bit arithmetic the product $111 \times 101 = 100011$ is rounded to 1.001×2^5. Throughout this paper, the symbols $\oplus$, $\ominus$, and $\otimes$ represent p-bit floating-point addition, subtraction, and multiplication with exact rounding. A number is said to be *expressible in p bits* if it can be expressed with a p-bit significand, *not* counting the sign bit or the exponent.

Algorithms for addition and multiplication of expansions follow. The (rather lengthy) proofs of all theorems are omitted, but are available in a full-length version of this paper. Theorems 3 and 6 are the key new results.

3.2 Addition

An important basic operation in all the algorithms for performing arithmetic with expansions is the addition of two p-bit values to form a nonoverlapping expansion (of length two). Two such algorithms follow.

Theorem 1 (Dekker [4]) *Let a and b be p-bit floating-point numbers such that $|a| \geq |b|$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$, where x is an approximation to $a + b$ and y represents the roundoff error in the calculation of x.* ■

FAST-TWO-SUM(a, b)
$x \Leftarrow a \oplus b$
$b_{\text{virtual}} \Leftarrow x \ominus a$
$y \Leftarrow b \ominus b_{\text{virtual}}$
return (x, y)

Note that the outputs x and y do *not* necessarily have the same sign. Two-term subtraction ("FAST-TWO-DIFF") is implemented by the sequence $x \Leftarrow a \ominus b$; $b_{\text{virtual}} \Leftarrow a \ominus x$; $y \Leftarrow b_{\text{virtual}} \ominus b$.

The difficulty with using FAST-TWO-SUM is the requirement that $|a| \geq |b|$. If the relative sizes of a and b are unknown, a comparison is required to order the addends before invoking FAST-TWO-SUM. In practice, it is faster on most processors to use the following algorithm.

Theorem 2 (Knuth [10]) *Let a and b be p-bit floating-point numbers, where $p \geq 3$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$.* ■

TWO-SUM(a, b)
$x \Leftarrow a \oplus b$
$b_{\text{virtual}} \Leftarrow x \ominus a$
$a_{\text{virtual}} \Leftarrow x \ominus b_{\text{virtual}}$
$b_{\text{roundoff}} \Leftarrow b \ominus b_{\text{virtual}}$
$a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$
$y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$
return (x, y)

Two-term subtraction ("TWO-DIFF") is implemented by the sequence $x \Leftarrow a \ominus b$; $b_{\text{virtual}} \Leftarrow a \ominus x$; $a_{\text{virtual}} \Leftarrow x \oplus b_{\text{virtual}}$; $b_{\text{roundoff}} \Leftarrow b_{\text{virtual}} \ominus b$; $a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$; $y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$.

Having established how to add two p-bit values, I turn to the topic of how to add two arbitrary precision values expressed as expansions. The algorithm LINEAR-EXPANSION-SUM below sums two expansions in linear time.

A complicating characteristic of the algorithm is that there may be spurious zero components scattered throughout the output expansion, even if no zeros are present in the input expansions. For instance, given the input expansions $1111 + 0.1001$ and $1100 + 0.1$, in four-bit arithmetic the output expansion is $11100 + 0 + 0 + 0.0001$. Interspersed zeros in input expansions do no harm except to slow down arithmetic, but this slowdown escalates quickly as expansions grow. It is important for LINEAR-EXPANSION-SUM and SCALE-EXPANSION to perform *zero elimination*, outputting a

[2] Note that this definition of *expansion* is slightly different from that used by Priest [12]; whereas Priest requires that the exponents of any two components of an expansion differ by at least p, no such requirement is made here.

[3] Formally, x and y are nonoverlapping if there exist integers r and s such that $x = r2^s$ and $|y| < 2^s$, or $y = r2^s$ and $|x| < 2^s$.

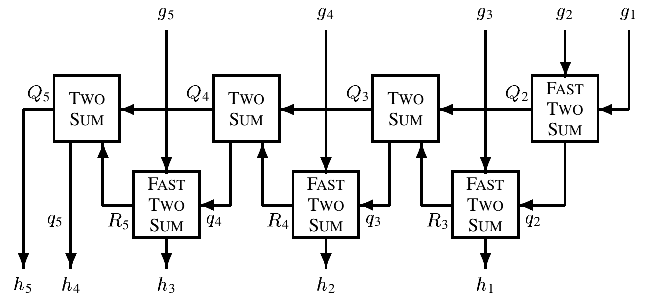

Figure 1: Operation of LINEAR-EXPANSION-SUM. The expansions g and h are illustrated with their most significant components on the left. $Q_i + q_i$ maintains an approximate running total. The FAST-TWO-SUM operations in the bottom row exist to clip a high-order bit off each q_i term, if necessary, before outputting it.

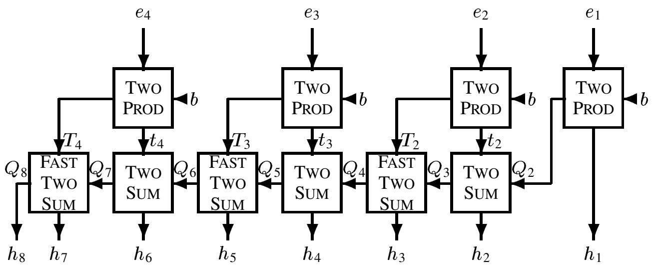

Figure 2: Operation of SCALE-EXPANSION. The expansions e and h are illustrated with their most significant components on the left.

component (and incrementing an array index) only if it is not zero. For simplicity, versions without zero elimination are presented here, but my implementations eliminate zeros.

Priest [12] presents a similar algorithm (for processors with arbitrary floating-point radix) that guarantees that the components of the output expansion overlap by at most one digit (i.e. one bit in binary arithmetic). An expensive renormalization step is required afterward to remove the overlap. By contrast, my algorithm always produces a nonoverlapping expansion, and no renormalization is needed. The algorithm takes advantage of binary arithmetic and exact rounding, but does not follow directly from Priest's results.

Theorem 3 *Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be nonoverlapping expansions of m and n p-bit components, respectively, where $p \geq 3$ and the components of both e and f are ordered by increasing magnitude. The following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{m+n} h_i = e + f$, where the components of h are also in order of increasing magnitude, except that any of the h_i may be zero.* ■

LINEAR-EXPANSION-SUM(e, f)
Merge e and f into a single sequence g, in order of nondecreasing magnitude
$(Q_2, q_2) \Leftarrow$ FAST-TWO-SUM(g_2, g_1)
for $i \Leftarrow 3$ **to** $m + n$
  $(R_i, h_{i-2}) \Leftarrow$ FAST-TWO-SUM(g_i, q_{i-1})
  $(Q_i, q_i) \Leftarrow$ TWO-SUM(Q_{i-1}, R_i)
$h_{m+n-1} \Leftarrow q_{m+n}$
$h_{m+n} \Leftarrow Q_{m+n}$
return h

$Q_i + q_i$ is an approximate sum of the first i terms of g; see Figure 1. It is possible to remove the FAST-TWO-SUM operation from the loop, yielding an algorithm that requires only six floating-point operations per iteration, but the preconditions for correct behavior are too complex to explain here.

3.3 Multiplication

The basic multiplication algorithm computes a nonoverlapping expansion equal to the product of two p-bit values. The multiplication is performed by splitting each value into two halves with half the precision, then performing four exact multiplications on these fragments. The trick is to find a way to split a floating-point value in two.

Theorem 4 (Dekker [4]) *Let a be a p-bit floating-point number, where $p \geq 3$. The following algorithm will produce a $\lfloor \frac{p}{2} \rfloor$-bit value a_{hi} and a nonoverlapping $(\lceil \frac{p}{2} \rceil - 1)$-bit value a_{lo} such that $|a_{\text{hi}}| \geq |a_{\text{lo}}|$ and $a = a_{\text{hi}} + a_{\text{lo}}$.* ■

SPLIT(a)
$c \Leftarrow (2^{\lceil p/2 \rceil} + 1) \otimes a$
$a_{\text{big}} \Leftarrow c \ominus a$
$a_{\text{hi}} \Leftarrow c \ominus a_{\text{big}}$
$a_{\text{lo}} \Leftarrow a \ominus a_{\text{hi}}$
return $(a_{\text{hi}}, a_{\text{lo}})$

The claim may seem absurd. After all, a_{hi} and a_{lo} have only $p - 1$ bits of significand between them; how can they carry all the information of a p-bit significand? The secret is hidden in the sign bit of a_{lo}. For instance, the seven-bit number 1001001 can be split into the three-bit terms 1010000 and -111. This property is fortunate, because even if p is odd, as it is in IEEE 754 double precision arithmetic, a can be split into two $\lfloor \frac{p}{2} \rfloor$-bit values.

Multiplication is performed by splitting a and b. The products $a_{\text{hi}} b_{\text{hi}}$, $a_{\text{lo}} b_{\text{hi}}$, $a_{\text{hi}} b_{\text{lo}}$, and $a_{\text{lo}} b_{\text{lo}}$ can each be computed exactly by the floating-point unit, producing four values. By subtracting them from $a \otimes b$ in a proper order, one is assured the subtractions are exact and the result is the roundoff error of computing $a \otimes b$. Dekker [4] attributes the following method to G. W. Veltkamp.

Theorem 5 (Veltkamp) *Let a and b be p-bit floating-point numbers, where $p \geq 4$. The following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$.* ■

TWO-PRODUCT(a, b)
$x \Leftarrow a \otimes b$
$(a_{\text{hi}}, a_{\text{lo}}) = $ SPLIT(a)
$(b_{\text{hi}}, b_{\text{lo}}) = $ SPLIT(b)
$err_1 \Leftarrow x \ominus (a_{\text{hi}} \otimes b_{\text{hi}})$
$err_2 \Leftarrow err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}})$
$err_3 \Leftarrow err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}})$
$y \Leftarrow (a_{\text{lo}} \otimes b_{\text{lo}}) \ominus err_3$
return (x, y)

The following algorithm, which multiplies an expansion by a floating-point value, is new.

Theorem 6 *Let $e = \sum_{i=1}^{m} e_i$ be an ordered nonoverlapping expansion of m p-bit components, and let b be a p-bit value where $p \geq 4$. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{2m} h_i = be$, where h is also ordered, except that any of the h_i may be zero. (See Figure 2.)* ■

```
SCALE-EXPANSION(e, b)
    (Q_2, h_1) ⇐ TWO-PRODUCT(e_1, b)
    for i ⇐ 2 to m
        (T_i, t_i) ⇐ TWO-PRODUCT(e_i, b)
        (Q_{2i−1}, h_{2i−2}) ⇐ TWO-SUM(Q_{2i−2}, t_i)
        (Q_{2i}, h_{2i−1}) ⇐ FAST-TWO-SUM(T_i, Q_{2i−1})
    h_{2m} ⇐ Q_{2m}
    return h
```

3.4 Approximation

The sign of an expansion can be identified by examining its largest component, but that component may be a poor approximation to the value of the whole expansion; it may carry as little as one bit of significance. Such a component may result, for instance, from cancellation during the subtraction of two nearly-equal expansions.

An APPROXIMATE procedure is defined that sums an expansion's components in order from smallest to largest. Because of the nonoverlapping property, APPROXIMATE produces an approximation having error less than the magnitude of the least significant bit of the approximation's significand.

4 Adaptive Precision Arithmetic

Exact arithmetic is expensive, and should be avoided when possible. The floating-point filter suggested by Fortune and Van Wyk [5], which tries to verify the correctness of the approximate result (using error analysis) before resorting to exact arithmetic, is quite effective. If the exact test is only needed occasionally, an application can be made robust at only a small cost in speed. One might hope to improve this idea by computing a sequence of several increasingly accurate results, testing each one in turn for accuracy. Alas, whenever an exact result is required, one suffers both the cost of the exact computation and the additional burden of computing several approximate results in advance. Fortunately, it is sometimes possible to use intermediate results as stepping stones to more accurate results; work already done is not discarded but is refined.

4.1 Making Arithmetic Adaptive

FAST-TWO-SUM, TWO-SUM, and TWO-PRODUCT each have the feature that they can be broken into two parts: Line 1, which computes an approximate result, and the remaining lines, which calculate the roundoff error. The latter, more expensive calculation can be delayed until it is needed, if it is ever needed at all. In this sense, these routines can be made *adaptive*, so that they only produce as much of the result as is needed. I describe here how to achieve the same effect with more general expressions.

Any expression composed of addition, subtraction, and multiplication operations can be calculated adaptively in a manner that defines a natural sequence of intermediate results whose accuracy it is appropriate to test. Such a sequence is most easily described by considering the tree associated with the expression, as in Figure 3(a). The leaves of this tree represent floating-point operands, and its internal nodes represent operations. Replace each node whose children are both leaves with the sum $x_i + y_i$, where x_i represents the approximate value of the subexpression, and y_i represents the roundoff error incurred while calculating x_i (Figure 3(b)), then expand the expression to form a polynomial.

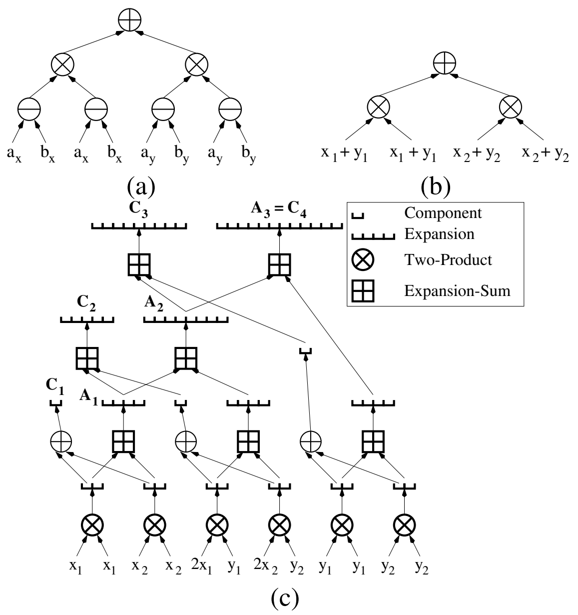

Figure 3: (a) Formula for the square of the distance between two points a and b. (b) The lowest subexpressions in the tree are expressed as the sum of an approximate value and a roundoff error. (c) An incremental adaptive method for evaluating the expression. The approximations C_1 through C_4 are generated and tested in turn. C_4 is exact.

In the expanded expression, the terms containing many occurrences of y variables are dominated by terms containing fewer occurrences. As an example, consider the expression $(a_x - b_x)^2 + (a_y - b_y)^2$ (Figure 3), which calculates the square of the distance between two points in the plane. Set $a_x - b_x = x_1 + y_1$ and $a_y - b_y = x_2 + y_2$. The resulting expression, expanded in full, is

$$(x_1^2 + x_2^2) + (2x_1y_1 + 2x_2y_2) + (y_1^2 + y_2^2).$$

It is significant that each y_i is small relative to its corresponding x_i. Exact rounding guarantees that $|y_i| \leq \epsilon |x_i|$, where $\epsilon = 2^{-p}$ is called the *machine epsilon*, and bounds the *relative error* $\text{error}(a \circledast b)/(a \circledast b)$ of any basic floating-point operation. In IEEE 754 double precision arithmetic, $\epsilon = 2^{-53}$; in single precision, $\epsilon = 2^{-24}$.

The expanded expression above can be divided into three parts, having magnitudes of $\mathcal{O}(1)$, $\mathcal{O}(\epsilon)$, and $\mathcal{O}(\epsilon^2)$, respectively. Denote these parts Φ_0, Φ_1, and Φ_2. More generally, for any expression expanded in this manner, let Φ_i be the sum of all products containing i of the y variables, so that Φ_i has magnitude $\mathcal{O}(\epsilon^i)$.

One can obtain an approximation A_j with error no larger than $\mathcal{O}(\epsilon^j)$ by exactly summing the first j terms, Φ_0 through Φ_{j-1}. The sequence $A_1, A_2, \ldots$ of increasingly accurate approximations can be computed incrementally; A_j is the exact sum of A_{j-1} and Φ_{j-1}. Members of this sequence are generated and tested until one is sufficiently accurate.

An improvement is based on the observation that one can obtain an approximation with error no larger than $\mathcal{O}(\epsilon^j)$ by adding (exactly) to A_{j-1} a correctional term that approximates Φ_{j-1} with ordinary floating-point arithmetic, to form a new approximation C_j, as illustrated in Figure 3(c). The correctional term reduces the error from $\mathcal{O}(\epsilon^{j-1})$ to $\mathcal{O}(\epsilon^j)$, so C_j is nearly as accurate as A_j but takes much less work to compute. This scheme reuses the work done in computing members of A, but does not reuse the (much cheaper) correctional terms. Note that C_1, the first value computed by this method, is an approximation to Φ_0; if C_1 is sufficiently accurate, it is unnecessary to use any exact arithmetic techniques at all. This first test is identical to Fortune and Van Wyk's floating-point filter.

This method does more work during each stage of the computation than the first method, but typically terminates one stage earlier. Although the use of correctional terms is slower when the exact result must be computed, it can cause a surprising improvement in other cases; for instance, the robust Delaunay tetrahedralization of points arrayed in a tilted grid (see Section 5.4) takes twice as long if the estimate C_2 is skipped in the orientation and incircle tests, because A_2 is much more expensive to produce.

The decomposition of an expression into an adaptive sequence as described above could be automated by an LN-like expression compiler, but for the predicates described in the next section, I have done the decomposition and written the code manually. Note that these ideas are not exclusively applicable to the multiple-term approach to arbitrary precision arithmetic. They can work with multiple-digit formats as well, though the details differ.

5 Predicate Implementations

5.1 Orientation and Incircle

Let a, b, c, and d be four points in the plane whose coordinates are machine-precision floating-point numbers. Define a procedure ORIENT2D(a, b, c) that returns a positive value if the points a, b, and c are arranged in counterclockwise order, a negative value if they are in clockwise order, and zero if they are collinear. A more common (but less symmetric) interpretation is that ORIENT2D returns a positive value if c lies to the left of the directed line ab; for this purpose the orientation test is used by many geometric algorithms.

Define also a procedure INCIRCLE(a, b, c, d) that returns a positive value if d lies inside the oriented circle abc. By *oriented circle*, I mean the unique (and possibly degenerate) circle through a, b, and c, with these points occurring in counterclockwise order about the circle. (If the points occur in clockwise order, INCIRCLE will reverse the sign of its output, as if the circle's exterior were its interior.) INCIRCLE returns zero if and only if all four points lie on a common circle.

These definitions extend to arbitrary dimensions. For instance, ORIENT3D(a, b, c, d) returns a positive value if d lies below the oriented plane passing through a, b, and c. By *oriented plane*, I mean that a, b, and c appear in counterclockwise order when viewed from above the plane.

In any dimension, the orientation and incircle tests may be implemented as matrix determinants. For example:

$$\text{ORIENT3D}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{1}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \tag{2}$$

$$\text{INCIRCLE}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} \tag{3}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix} \tag{4}$$

These formulae generalize to other dimensions in the obvious way. Expressions 1 and 2 can be shown to be equivalent by simple algebraic transformations, as can Expressions 3 and 4. These equivalences are unsurprising because one expects the results of any orientation or incircle test not to change if all the points undergo an identical translation in the plane. Expression 2, for instance, follows from Expression 1 by translating each point by $-d$.

For exact computation, the choice between Expressions 1 and 2, or between 3 and 4, is not straightforward. Expression 2 takes roughly 25% more time to compute in exact arithmetic, and Expression 4 takes about 30% more time than Expression 3. The disparity likely increases in higher dimensions. Nevertheless, the mechanics of error estimation turn the tide in the other direction. Important as a fast exact test is, it is equally important to avoid exact tests whenever possible. Expressions 2 and 4 tend to have smaller errors (and correspondingly smaller error estimates) because their errors are a function of the relative coordinates of the points, whereas the errors of Expressions 1 and 3 are a function of the absolute coordinates of the points.

In most geometric applications, the points that serve as parameters to geometric tests tend to be close to each other. Commonly, their absolute coordinates are much larger than the distances between them. By translating the points so they lie near the origin, working precision is freed for the subsequent calculations. Hence, the errors and error bounds

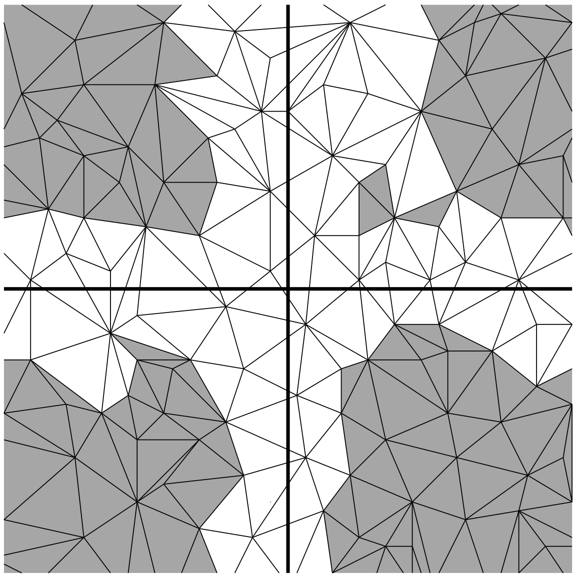

Figure 4: Shaded triangles can be translated to the origin without incurring roundoff error. In most triangulations, such triangles are the common case.

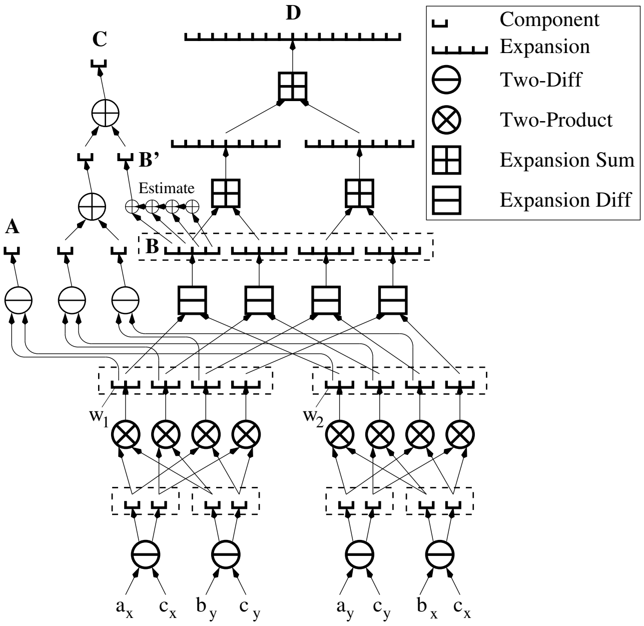

Figure 5: Adaptive calculations used by the 2D orientation test. Dashed boxes represent nodes in the original expression tree.

for Expressions 2 and 4 are generally much smaller than for Expressions 1 and 3. Furthermore, the translation can often be done without roundoff error. Figure 4 demonstrates a toy problem: suppose ORIENT2D is used to find the orientation of each triangle in a triangulation. A well-known property of floating-point arithmetic is that if two p-bit floating-point values have the same sign and differ by at most a factor of two, their difference is expressible in p bits. Hence, any shaded triangle can be translated so that one of its vertices lies at the origin without roundoff error; the white triangles may or may not suffer from roundoff during such translation. If the complete triangulation is much larger than the portion illustrated, only a small proportion of the triangles (those near a coordinate axis) can suffer roundoff. Because exact translation is the common case, my adaptive geometric predicates test for and exploit this case.

Once a determinant has been chosen for evaluation, there are several methods to evaluate it. A few are surveyed by Fortune and Van Wyk [5]; only their conclusion is repeated here. The cheapest method of evaluating the determinant of a 5×5 or smaller matrix seems to be dynamic programming applied to cofactor expansion. Evaluate the $\binom{d}{2}$ determinants of all 2×2 minors of the first two columns, then the $\binom{d}{3}$ determinants of all 3×3 minors of the first two columns, and so on. All four of my predicates use this method.

5.2 ORIENT2D

My implementation of ORIENT2D computes a sequence of up to four results (labeled A through D) as illustrated in Figure 5. The exact result D may be as long as sixteen components, but zero elimination is used, so a length of two to six components is more common in practice.

A, B, and C are logical places to test the accuracy of the result before continuing. In most applications, the majority of calls to ORIENT2D will end with the approximation A. Although the four-component expansion B, like A, has an error of $\mathcal{O}(\epsilon)$, it is a likely stopping point because B is exact if the four subtractions at the bottom of the expression tree are performed without roundoff error (corresponding to the shaded triangles in Figure 4). Because this is the common case, ORIENT2D explicitly tests whether all the roundoff terms are zero. The corrected estimate C has an error bound of $\mathcal{O}(\epsilon^2)$. If C is not sufficiently accurate, the exact determinant D is computed.

There are two interesting features of this test, both of which arise because only the sign of the determinant is needed. First, the correctional term added to B to form C is not added exactly; instead, the APPROXIMATE procedure of Section 3.4 finds an approximation B′ of B, and the correctional term is added to B′ with the possibility of roundoff error. The consequent errors may be of magnitude $\mathcal{O}(\epsilon \mathrm{B})$, and would normally preclude obtaining an error bound of $\mathcal{O}(\epsilon^2)$. However, the sign of the determinant is only questionable if B is of magnitude $\mathcal{O}(\epsilon)$, so an $\mathcal{O}(\epsilon^2)$ error bound for C can be established.

The second interesting feature is that, if C is not sufficiently accurate, no more approximations are computed before computing the exact determinant. To understand why, consider three collinear points; the determinant defined by these points is zero. If a coordinate of one of these points is perturbed by the least significant bit of its significand, the determinant typically increases to $\mathcal{O}(\epsilon)$. Hence, one might guess that when a determinant is no larger than $\mathcal{O}(\epsilon^2)$, it is probably zero. This intuition seems to hold in practice for all four predicates

Double precision ORIENT2D timings in microseconds			
Points	Uniform	Geometric	Nearly
Method	Random	Random	Collinear
Approximate (2)	0.15	0.15	0.16
Exact (1)	6.56	6.89	6.31
Exact (2)	8.35	8.48	8.13
Exact (1), MPFUN	92.85	94.03	84.97
Adaptive A (2), approx.	0.28	0.27	0.22
Adaptive B (2)			1.89
Adaptive C (2)			2.14
Adaptive D (2), exact			8.35
LN adaptive (2), approx.	0.32	n/a	
LN adaptive (2), exact		n/a	4.43

Table 1: Timings for ORIENT2D on a DEC 3000/700 with a 225 MHz Alpha processor. All determinants use the 2D version of either Expression 1 or the more stable Expression 2 as indicated. Timings for the adaptive tests are categorized according to which result was the last generated. Timings of Bailey's MPFUN package and Fortune and Van Wyk's LN package are included for comparison.

considered herein, on both random and "practical" point sets. Determinants that don't stop with approximation C are nearly always zero.

The error bound for A is $(3\epsilon + 16\epsilon^2) \otimes (|w_1| \oplus |w_2|)$, where w_1 and w_2 are as indicated in Figure 5. This error bound has the pleasing property that it is zero in the common case that all three input points lie on a horizontal or vertical line. Hence, although ORIENT2D usually resorts to exact arithmetic when given collinear input points, it only performs the approximate test in the two cases that occur most commonly in practice.

Table 1 lists timings for ORIENT2D, given random inputs. Observe that the adaptive test, when it stops at the approximate result A, takes nearly twice as long as the approximate test because of the need to compute an error bound. The table includes a comparison with Bailey's MPFUN [2], chosen because it is the fastest portable and freely available arbitrary precision package I know of. ORIENT2D coded with my (nonadaptive) algorithms is roughly thirteen times faster than ORIENT2D coded with MPFUN.

Also included is a comparison with an orientation predicate for 53-bit integer inputs, created by Fortune and Van Wyk's LN. The LN-generated orientation predicate is quite fast because it takes advantage of the fact that it is restricted to bounded integer inputs. My exact tests cost less than twice as much as LN's; this seems like a reasonable price to pay for the ability to handle arbitrary exponents in the input.

These timings are not the whole story; LN's static error estimate is typically much larger than the runtime error estimate used for adaptive stage A, and LN uses only two stages of adaptivity, so the LN-generated predicates are slower in some applications, as Section 5.4 will demonstrate. (It is significant that for 53-bit integer inputs, my multiple-stage predicates rarely pass stage B because the initial translation is usually done without roundoff error; hence, the LN-generated ORIENT2D often takes more than twice as long to produce an exact result.) It must be emphasized, however, that these are not inherent differences between LN's multiple-digit integer approach and my multiple-term floating-point approach; LN could, in principle, employ the same runtime error estimate and a similar multiple-stage adaptivity scheme.

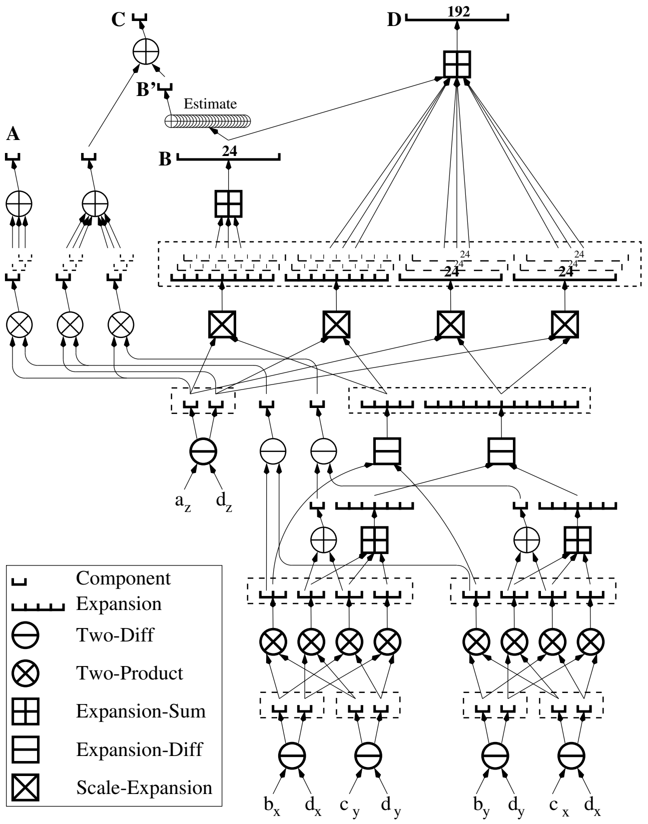

Figure 6: Adaptive calculations used by the 3D orientation test. Bold numbers indicate the length of an expansion. Only part of the expression tree is shown; two of the three cofactors are omitted, but their results appear as dashed components and expansions.

5.3 ORIENT3D, INCIRCLE, and INSPHERE

Figure 6 illustrates the implementation of ORIENT3D. As with ORIENT2D, A is the standard floating-point result, B is exact if the subtractions at the bottom of the tree incur no roundoff, C represents a drop in the error bound from $\mathcal{O}(\epsilon)$ to $\mathcal{O}(\epsilon^2)$, and D is the exact determinant. The error bounds are zero if all four input points share the same x, y, or z-coordinate, so only the approximate test is needed in the most common cases of coplanarity.

Table 2 lists timings for ORIENT3D, INCIRCLE, and INSPHERE, given random inputs. In each case, the error bound for A increases the amount of time required to perform the approximate test in the adaptive case by a factor of 2 to 2.5. The gap between my exact algorithms and MPFUN is smaller than in the case of ORIENT2D, but is still a factor of 3 to 7.5.

INCIRCLE is implemented similarly to ORIENT3D, as the determinants are similar. The implementation of INSPHERE differs from the other three tests in that, due to programmer laziness, D is not computed incrementally from B; rather, if C is not accurate enough, D is computed from scratch.

Double precision Orient3D timings in microseconds			
Method / Points	Uniform Random	Geometric Random	Nearly Coplanar
Approximate (2)	0.25	0.25	0.25
Exact (1)	33.30	38.54	32.90
Exact (2)	42.69	48.21	42.41
Exact (1), MPFUN	260.51	262.08	246.64
Adaptive A (2), approx.	0.61	0.60	0.62
Adaptive B (2)			12.98
Adaptive C (2)			15.59
Adaptive D (2), exact			27.29
LN adaptive (2), approx.	0.85	n/a	
LN adaptive (2), exact		n/a	18.11

Double precision InCircle timings in microseconds			
Method / Points	Uniform Random	Geometric Random	Nearly Cocirc.
Approximate (4)	0.31	0.28	0.30
Exact (3)	71.66	83.01	75.34
Exact (4)	91.71	118.30	104.44
Exact (3), MPFUN	350.77	343.61	348.55
Adaptive A (4), approx.	0.64	0.59	0.64
Adaptive B (4)			44.56
Adaptive C (4)			48.80
Adaptive D (4), exact			78.06
LN adaptive (4), approx.	1.33	n/a	
LN adaptive (4), exact		n/a	32.44

Double precision InSphere timings in microseconds			
Method / Points	Uniform Random	Geometric Random	Nearly Cospher.
Approximate (4)	0.93	0.95	0.93
Exact (3)	324.22	378.94	347.16
Exact (4)	374.59	480.28	414.13
Exact (3), MPFUN	1,017.56	1,019.89	1,059.87
Adaptive A (4), approx.	2.13	2.14	2.14
Adaptive B (4)			166.21
Adaptive C (4)			171.74
Adaptive D (4), exact			463.96
LN adaptive (4), approx.	2.35	n/a	
LN adaptive (4), exact		n/a	116.74

Table 2: Timings for Orient3D, InCircle, and InSphere on a DEC 3000/700. All determinants are Expression 1 or 3, or the more stable Expression 2 or 4, as indicated.

Fortunately, C is usually accurate enough.

5.4 Triangulation

To evaluate the effectiveness of the adaptive tests in applications, I tested them in two of my Delaunay triangulation codes. Triangle [14] is a 2D Delaunay triangulator and mesh generator, publicly available from Netlib, that uses a divide-and-conquer algorithm [11, 8]. Pyramid is a 3D Delaunay tetrahedralizer that uses an incremental algorithm [15]. For both 2D and 3D, three types of inputs were tested: uniform random points, points lying (approximately) on the boundary of a circle or sphere, and a square or cubic grid of lattice points, tilted so as not to be aligned with the coordinate axes. The latter two were chosen for their nastiness. The lattices have been tilted using approximate arithmetic, so they are not perfectly cubical, and the exponents of their coordinates vary

2D divide-and-conquer Delaunay triangulation			
	Uniform Random	Perimeter of Circle	Tilted Grid
Input points	1,000,000	1,000,000	1,000,000
Orient2D calls			
Adaptive A, approx.	9,497,314	6,291,742	9,318,610
Adaptive B			121,081
Adaptive C			118
Adaptive D, exact			3
Average time, μs	0.32	0.38	0.33
LN approximate	9,497,314	2,112,284	n/a
LN exact		4,179,458	n/a
LN average time, μs	0.35	3.16	n/a
InCircle calls			
Adaptive A, approx.	7,596,885	3,970,796	7,201,317
Adaptive B		50,551	176,470
Adaptive C		120	47
Adaptive D, exact			4
Average time, μs	0.65	1.11	1.67
LN approximate	6,077,062	0	n/a
LN exact	1,519,823	4,021,467	n/a
LN average time, μs	7.36	32.78	n/a
Program running time, seconds			
Approximate version	57.3	59.9	48.3
Robust version	61.7	64.7	62.2
LN robust version	116.0	214.6	n/a

Table 3: Statistics for 2D divide-and-conquer Delaunay triangulation of several point sets.

3D incremental Delaunay tetrahedralization			
	Uniform Random	Surface of Sphere	Tilted Grid
Input points	10,000	10,000	10,000
Orient3D counts			
Adaptive A, approx.	2,735,668	1,935,978	5,542,567
Adaptive B			602,344
Adaptive C			1,267,423
Adaptive D, exact			28,185
Average time, μs	0.72	0.72	4.12
LN approximate	2,735,668	1,935,920	n/a
LN exact		58	n/a
LN average time, μs	0.99	1.00	n/a
InSphere counts			
Adaptive A, approx.	439,090	122,273	3,080,312
Adaptive B		180,383	267,162
Adaptive C		1,667	548,063
Adaptive D, exact			
Average time, μs	2.23	96.45	48.12
LN approximate	438,194	104,616	n/a
LN exact	896	199,707	n/a
LN average time, μs	2.50	70.82	n/a
Program running time, seconds			
Approximate version	4.3	3.0	∞
Robust version	5.8	34.1	108.5
LN robust version	6.5	30.5	n/a

Table 4: Statistics for 3D incremental Delaunay tetrahedralization of several point sets. The approximate code failed to terminate on the tilted grid input.

enough that LN cannot be used. (I could have used perfect lattices with 53-bit integer coordinates, but ORIENT3D and INSPHERE would never pass stage B; the perturbed lattices occasionally force the predicates into stage C or D.)

The results for 2D, outlined in Table 3, indicate that the four-stage predicates add about 8% to the total running time for randomly distributed input points, mainly because of the error bound tests. For the more difficult point sets, the penalty may be as great as 30%. Of course, this penalty applies to precisely the point sets that are most likely to cause difficulties when exact arithmetic is not available.

The results for 3D, outlined in Table 4, are somewhat less pleasing. The four-stage predicates add about 35% to the total running time for randomly distributed input points; for points distributed approximately on the surface of a sphere, the penalty is a factor of eleven. Ominously, however, the penalty for the tilted grid is uncertain, because the tetrahedralization program using approximate arithmetic failed to terminate. A debugger revealed that the point location routine was stuck in an infinite loop because a geometric inconsistency had been introduced into the mesh due to roundoff error. Robust arithmetic is not always slower after all.

6 Conclusions

As Priest points out, multiple-term algorithms can be used to implement extended (but finite) precision arithmetic as well as exact arithmetic; simply compress and then truncate each result to a fixed number of components. Perhaps the greatest potential of these algorithms is in providing a fast and simple way to extend slightly the precision of critical variables in numerical algorithms. Hence, it would not be difficult to provide a routine that quickly computes the intersection point of two segments with double precision endpoints, correctly rounded to a double precision result. Speed considerations may make it untenable to accomplish this by calling a standard extended precision library. The techniques Priest and I have developed are simple enough to be coded directly in numerical algorithms, avoiding function call overhead and conversion costs.

A useful tool in coding such algorithms would be an expression compiler similar to Fortune and Van Wyk's [5], which converts an expression into exact arithmetic code, complete with error bound derivation and floating-point filters. Such a tool might even be able to automate the process of breaking an expression into adaptive stages as described in Section 4.

It might be fruitful to explore whether the methods described by Clarkson [3] and Avnaim et al. [1] can be extended by fast multiprecision methods to handle arbitrary double precision floating-point inputs. One could certainly relax their constraints on the bit complexity of the inputs; for instance, the method of Avnaim et al. could be made to perform the INSPHERE test on 64-bit inputs using expansions of length three. Unfortunately, it is not obvious how to adapt these integer-based techniques to inputs with wildly differing exponents. It is also not clear whether such hybrid algorithms would be faster than straightforward adaptivity. Nevertheless, Clarkson's approach looks promising for larger determinants. Although my methods work well for small determinants, they are unlikely to work well for sizes much larger than 5×5. Even if one uses Gaussian elimination rather than cofactor expansion (an important adjustment for larger matrices [5, 9]), the adaptivity technique does not scale well with determinants, because of the large number of terms in the expanded polynomial. Clarkson's technique may be the only economical approach for matrices larger than 10×10.

Whether or not these issues are resolved in the near future, researchers can make use today of tests for orientation and incircle in two and three dimensions that are correct, fast, and immediately applicable to double precision floating-point inputs. I invite working computational geometers to try my code in their implementations, and I hope that it will save them from worrying about robustness so they may concentrate on geometry.

References

[1] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic*. 1995.

[2] David H. Bailey. *A Portable High Performance Multiprecision Package*. Technical Report RNR-90-022, NASA Ames Research Center, May 1993.

[3] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation*. 33rd Annual Symposium on Foundations of Computer Science, pages 387–395, 1992.

[4] T. J. Dekker. *A Floating-Point Technique for Extending the Available Precision*. Numerische Mathematik **18**:224–242, 1971.

[5] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry*. Ninth Annual Symposium on Computational Geometry, pages 163–172, May 1993.

[6] ———. *Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry*. To appear in Transactions on Mathematical Software, 1996.

[7] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys **23**(1):5–48, March 1991.

[8] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams*. ACM Transactions on Graphics **4**(2):74–123, April 1985.

[9] Michael Karasick, Derek Lieber, and Lee R. Nackman. *Efficient Delaunay Triangulation Using Rational Arithmetic*. ACM Transactions on Graphics **10**(1):71–91, January 1991.

[10] Donald Ervin Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, second edition, volume 2. Addison Wesley, 1981.

[11] D. T. Lee and B. J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation*. Int. J. Comput. Inf. Sci. **9**:219–242, 1980.

[12] Douglas M. Priest. *Algorithms for Arbitrary Precision Floating Point Arithmetic*. Tenth Symposium on Computer Arithmetic, pages 132–143, 1991.

[13] ———. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. thesis, Department of Mathematics, University of California at Berkeley, November 1992.

[14] Jonathan Richard Shewchuk. *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*. First Workshop on Applied Computational Geometry. Association for Computing Machinery, May 1996.

[15] David F. Watson. *Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes*. Computer Journal **24**:167–172, 1981.