Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates

Jonathan Richard Shewchuk

October 1, 1997

CMU-CS-96-140R

From *Discrete & Computational Geometry* **18**(3):305–363, October 1997.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Exact computer arithmetic has a variety of uses, including the robust implementation of geometric algorithms. This article has three purposes. The first is to offer fast software-level algorithms for exact addition and multiplication of arbitrary precision floating-point values. The second is to propose a technique for adaptive precision arithmetic that can often speed these algorithms when one wishes to perform multiprecision calculations that do not always require exact arithmetic, but must satisfy some error bound. The third is to use these techniques to develop implementations of several common geometric calculations whose required degree of accuracy depends on their inputs. These robust geometric predicates 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 complying with the IEEE 754 standard. The inputs to the predicates may be arbitrary single or double precision floating-point numbers. C code is publicly available for the 2D and 3D orientation and incircle tests, and robust Delaunay triangulation using these tests. Timings of the implementations demonstrate their effectiveness.

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 CMS-9318163. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either express or implied, of NSERC, NSF, or the U.S. Government.

Keywords: arbitrary precision floating-point arithmetic, computational geometry, geometric robustness, orientation test, incircle test, Delaunay triangulation

Contents

About this Report

An electronic copy of this report, and the software described herein, can be obtained through the Web page http://www.cs.cmu.edu/∼quake/robust.html.

Many thanks to Steven Fortune, Douglas Priest, and Christopher Van Wyk, who each provided comments on a draft of this paper, and whose papers provided the foundations for this research. Steven Fortune also provided LN-generated predicates for timing comparisons, and unwittingly sparked this research in mid-1994 with a few brief email responses. Thanks also to David O'Hallaron, James Stichnoth, and Daniel Tunkelang for their comments.

1 Introduction

Software libraries for arbitrary precision floating-point arithmetic can be used to accurately perform many error-prone or ill-conditioned computations that would be infeasible using only hardware-supported approximate arithmetic. Some of these computations have accuracy requirements that vary with their input. For instance, consider the problem of finding the center of a circle, given three points that lie on the circle. Normally, hardware precision arithmetic will suffice, but if the input points are nearly collinear, the problem is ill-conditioned and the approximate calculation may yield a wildly inaccurate result or a division by zero. Alternatively, an exact arithmetic library can be used and will yield a correct result, but exact arithmetic is slow; one would rather use it only when one really needs to.

This article presents two techniques for writing fast implementations of extended precision calculations like these, and demonstrates them with implementations of four commonly used geometric predicates. The first technique is a suite of algorithms, several of them new, for performing arbitrary precision arithmetic. 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 and some numerical methods, where much benefit can be realized from a modest increase in precision. The second technique is a way to modify these algorithms so that they compute their result adaptively; they are quick in most circumstances, but are still slow when their results are prone to have high relative error. A third subject of this article is a demonstration of these techniques with implementations and performance measurements of four commonly used geometric predicates. An elaboration of each of these three topics follows.

Methods of simulating exact arithmetic in software can be classified by several characteristics. Some exact arithmetic libraries operate on integers or fixed-point numbers, while others operate on floating-point numbers. To represent a number, the former libraries store a significand of arbitrary length; the latter store an exponent as well. Some libraries use the hardware's integer arithmetic units, whereas others use the floating-point units. Oddly, the decision to use integers or floating-point numbers internally is orthogonal to the type of number being represented. It was once the norm to use integer arithmetic to build extended precision floating-point libraries, especially when floating-point hardware was uncommon and differed between computer models. Times have changed, and modern architectures are highly optimized for floating-point performance; on many processors, floating-point arithmetic is faster than integer arithmetic. The trend is reversing for software libraries as well, and there are several proposals to use floating-point arithmetic to perform extended-precision integer calculations. Fortune and Van Wyk [12, 11], Clarkson [6], and Avnaim, Boissonnat, Devillers, Preparata, and Yvinec [1] have described algorithms of this kind, designed to attack the same computational geometry robustness problems considered later in this article. These algorithms are surveyed in Section 4.1.

Another differentiating feature of multiprecision libraries is whether they use multiple exponents. 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-component* format, wherein a number is expressed as a sum of ordinary floating-point words, each with its own significand and exponent [23, 7, 19]. This approach has the advantage that the result of an addition like $2^{300} + 2^{-300}$ (which may well arise in calculations like the geometric predicates discussed in Section 4.2) 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. On the other hand, the multiple-digit approach can more compactly represent most numbers, because only one exponent is stored. (MPFUN sacrifices this compactness to take advantage of floating-point hardware; the exponent of each digit is unused.) More pertinent is the difference in speed, discussed briefly in Section 2.1.

The algorithms described herein use floating-point hardware to perform extended precision floating-point arithmetic, using the multiple-component approach. These algorithms, described in Section 2, work under the assumption that hardware arithmetic is performed in radix two with exact rounding. This assumption holds on processors compliant with the IEEE 754 floating-point standard. Proofs of the correctness of all algorithms are given.

The methods herein are closely related to, and occasionally taken directly from, methods developed by Priest [23, 24], but are faster. The improvement in speed arises partly because Priest's algorithms run on a wide variety of floating-point architectures, with different radices and rounding behavior, whereas mine are limited to and optimized for radix two with exact rounding. This specialization is justified by the wide acceptance of the IEEE 754 standard. My algorithms also benefit from a relaxation of Priest's normalization requirement, which is less strict than the normalization required by multiple-digit algorithms, but is nonetheless time-consuming to enforce.

I demonstrate these methods with publicly available code that performs the two-dimensional and three-dimensional orientation and incircle tests, calculations that commonly arise in computational geometry. 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 [14]. 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 [16, 11]. For these reasons, computational geometry is an important arena for evaluating extended precision arithmetic schemes.

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 [11] 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, approximations reuse a previous, less accurate computation when it is economical to do so. Procedures thus designed are adaptive; they refine their results until they are certain of the correctness of their answer. The technique is not limited to computational geometry, nor is it limited to finding signs of expressions; it can be employed in any calculation where the required degree of accuracy varies. This adaptive approach is described in Section 3, and its application to the orientation and incircle tests is described in Section 4.

Readers who wish to use these predicates in their own applications are encouraged to download them and try them out. However, be certain to read Section 5, which covers two important issues that must be considered to ensure the correctness of the implementation: your processor's floating-point behavior and your compiler's optimization behavior. Furthermore, be aware that exact arithmetic is not a panacea for all robustness woes; its uses and limitations are discussed in Section 4.1. Exact arithmetic can make robust many algorithms that take geometric input and return purely combinatorial output; for instance, a fully robust convex hull implementation can be produced with recourse only to an exact orientation test. However, in algorithms that construct new geometric objects, exact arithmetic is sometimes constrained by its cost and its inability to represent arbitrary irrational numbers.

A few words are appropriate to describe some of the motivation for pursuing robust predicates for floating-point, rather than integer, operands. One might argue that real-valued input to a geometric program can be scaled and approximated in integer form. Indeed, there are few geometric problems that truly require the range of magnitude that floating-point storage provides, and integer formats had a clear speed advantage over floating-point formats for small-scale exact computation prior to the present research. The best argument for exact floating-point libraries in computational geometry, besides convenience, is the fact that many existing geometric programs already use floating-point numbers internally, and it is easier to replace their geometric predicates with robust floating-point versions than to retrofit the programs to use integers throughout. Online algorithms present another argument, because they are not always compatible with the scaled-input approach. One cannot always know in advance what resolution will be required, and repeated rescalings may be necessary to support an internal integer format when the inputs are real and unpredictable. In any case, I hope that this research will make it easier for programmers to choose between integer and floating-point arithmetic as they prefer.

2 Arbitrary Precision Floating-Point Arithmetic

2.1 Background

Most modern processors support floating-point numbers of the form $\pm$*significand* $\times\ 2^{\mathit{exponent}}$. The significand is a p-bit binary number of the form $b.bbb\ldots$, where each b denotes a single bit; one additional bit represents the sign. This article 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 whose exponents fall within a circumscribed range. The four predicates implemented for this article will not overflow nor underflow if their inputs have exponents in the range $[-142, 201]$ and IEEE 754 double precision arithmetic is used.) Floating-point values are generally *normalized*, which means that if a value is not zero, then its most significant bit is set to one, and the exponent adjusted accordingly. For example, in four-bit arithmetic, binary 1101 (decimal 13) is represented as 1.101×2^3. See the survey by Goldberg [13] for a detailed explanation of floating-point storage formats, particularly the IEEE 754 standard.

Exact arithmetic often produces values that require more than p bits to store. For the algorithms herein, each arbitrary precision value is expressed as an *expansion*[1] $x = x_n + \cdots + x_2 + x_1$, 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.[2] 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 component with largest magnitude).

Two floating-point values x and y are *adjacent* if they overlap, if x overlaps $2y$, or if $2x$ overlaps y. For instance, 1100 is adjacent to 11, but 1000 is not. An expansion is *nonadjacent* if no two of its components are adjacent. Surprisingly, any floating-point value has a corresponding nonadjacent expansion; for instance, 11111 may appear at first not to be representable as a nonoverlapping expansion of one-bit components, but

[1] Note that this definition of *expansion* is slightly different from that used by Priest [23]; whereas Priest requires that the exponents of any two components of the expansion differ by at least p, no such requirement is made here.

[2] 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 $|x| < 2^s$.

consider the expansion $100000 + -1$. The trick is to use the sign bit of each component to separate it from its larger neighbor. We will later see algorithms in which nonadjacent expansions arise naturally.

Multiple-component algorithms (based on the expansions defined above) can be faster than multiple-digit algorithms because the latter require expensive normalization of results to fixed digit positions, whereas multiple-component algorithms can allow the boundaries between components 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 any ordinary floating-point number *is* an expansion of length one. Conversion overhead can account for a significant part of the cost of small extended precision computations.

The central conceptual difference between standard multiple-digit algorithms and the multiple-component 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. For instance, in four-bit arithmetic the product $111 \times 101 = 100011$ is rounded to 1.001×2^5. If a value falls precisely halfway between two consecutive p-bit values, a tiebreaking rule determines the result. Two possibilities are the round-to-even rule, which specifies that the value should be rounded to the nearest p-bit value with an even significand, and the round-toward-zero rule. In four-bit arithmetic, 10011 is rounded to 1.010×2^4 under the round-to-even rule, and to 1.001×2^4 under the round-toward-zero rule. The IEEE 754 standard specifies round-to-even tiebreaking as a default. Throughout this article, the symbols $\oplus$, $\ominus$, and $\otimes$ represent p-bit floating-point addition, subtraction, and multiplication with exact rounding. Due to roundoff, these operators lack several desirable arithmetic properties. Associativity is an example; in four-bit arithmetic, $(1000 \oplus 0.011) \oplus 0.011 = 1000$, but $1000 \oplus (0.011 \oplus 0.011) = 1001$. A list of reliable identities for floating-point arithmetic is given by Knuth [17].

Roundoff is often analyzed in terms of *ulps*, or "units in the last place." An ulp is the effective magnitude of the low-order (pth) bit of a p-bit significand. An ulp is defined relative to a specific floating point value; I shall use $\text{ulp}(a)$ to denote this quantity. For instance, in four-bit arithmetic, $\text{ulp}(-1100) = 1$, and $\text{ulp}(1) = 0.001$.

Another useful notation is $\text{err}(a \circledast b)$, which denotes the roundoff error incurred by using a p-bit floating-point operation $\circledast$ to approximate a real operation $*$ (addition, subtraction, multiplication, or division) on the operands a and b. Note that whereas ulp is an unsigned quantity, err is signed. For any basic operation, $a \circledast b = a * b + \text{err}(a \circledast b)$, and exact rounding guarantees that $|\text{err}(a \circledast b)| \leq \frac{1}{2}\text{ulp}(a \circledast b)$.

In the pages that follow, various properties of floating-point arithmetic are proven, and algorithms for manipulating expansions are developed based on these properties. Throughout, binary and decimal numbers are intermixed; the base should be apparent from context. 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. I will occasionally refer to the *magnitude of a bit*, defined relative to a specific number; for instance, the magnitude of the second nonzero bit of binary -1110 is four. The remainder of this section is quite technical; the reader may wish to skip the proofs on a first reading. The key new results are Theorems 13, 19, and 24, which provide algorithms for summing and scaling expansions.

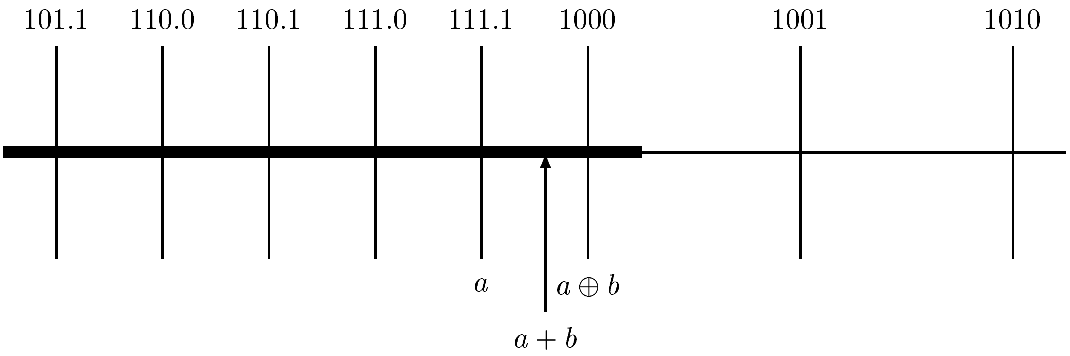

Figure 1: Demonstration of the first two lemmata. Vertical lines represent four-bit floating-point values. The roundoff error is the distance between $a + b$ and $a \oplus b$. Lemma 1 states that the error cannot be larger than $|b|$. Lemma 3(b) states that if $|a + b| \leq 2^i(2^{p+1} + 1)$ (for $i = -2$ and $p = 4$, this means that $a + b$ falls into the darkened region), then the error is no greater than 2^i. This lemma is useful when a computed value falls close to a power of two.

2.2 Properties of Binary Arithmetic

Exact rounding guarantees that $|\text{err}(a \circledast b)| \leq \frac{1}{2}\text{ulp}(a \circledast b)$, but one can sometimes find a smaller bound for the roundoff error, as evidenced by the two lemmata below. The first lemma is useful when one operand is much smaller than the other, and the second is useful when the sum is close to a power of two. For Lemmata 1 through 5, let a and b be p-bit floating-point numbers.

Lemma 1 *Let* $a \oplus b = a + b + \text{err}(a \oplus b)$. *The roundoff error* $|\text{err}(a \oplus b)|$ *is no larger than* $|a|$ *or* $|b|$. *(An analogous result holds for subtraction.)*

Proof: Assume without loss of generality that $|a| \geq |b|$. The sum $a \oplus b$ is the p-bit floating-point number closest to $a + b$. But a is a p-bit floating-point number, so $|\text{err}(a \oplus b)| \leq |b| \leq |a|$. (See Figure 1.) ■

Corollary 2 *The roundoff error* $\text{err}(a \oplus b)$ *can be expressed with a* p*-bit significand.*

Proof: Assume without loss of generality that $|a| \geq |b|$. Clearly, the least significant nonzero bit of $\text{err}(a \oplus b)$ is no smaller in magnitude than $\text{ulp}(b)$. By Lemma 1, $|\text{err}(a \oplus b)| \leq |b|$; hence, the significand of $\text{err}(a \oplus b)$ is no longer than that of b. It follows that $\text{err}(a \oplus b)$ is expressible in p bits.

Lemma 3 *For any basic floating-point operation* $*$, *let* $a \circledast b = a * b + \text{err}(a \circledast b)$. *Then:*

(a) If $|\text{err}(a \circledast b)| \geq 2^i$ *for some integer* i*, then* $|a * b| \geq 2^i(2^p + 1)$.

(b) If $|\text{err}(a \circledast b)| > 2^i$ *for some integer* i*, then* $|a * b| > 2^i(2^{p+1} + 1)$.

$$\begin{array}{rcrrrr} a & = & 1 & 1 & 0 & 1 \\ b & = & 1 & 0 & 1 & 0 \\ \hline a-b & = & & & 1 & 1 \end{array} \qquad \begin{array}{rcrrrrrl} a & = & 1 & 0 & 0 & 1 & & \times 2^1 \\ b & = & & 1 & 0 & 0 & 1 & \\ \hline a-b & = & & 1 & 0 & 0 & 1 & \end{array}$$

Figure 2: Two demonstrations of Lemma 5.

Proof:

(a) The numbers $2^i(2^p), 2^i(2^p-1), 2^i(2^p-2), \ldots, 0$ are all expressible in p bits. Any value $|a * b| < 2^i(2^p+1)$ is within a distance less than 2^i from one of these numbers.

(b) The numbers $2^i(2^{p+1}), 2^i(2^{p+1}-2), 2^i(2^{p+1}-4), \ldots, 0$ are all expressible in p bits. Any value $|a * b| \leq 2^i(2^{p+1}+1)$ is within a distance of 2^i from one of these numbers. (See Figure 1.) ∎

The next two lemmata identify special cases for which computer arithmetic is exact. The first shows that addition and subtraction are exact if the result has smaller magnitude than the operands.

Lemma 4 *Suppose that $|a+b| \leq |a|$ and $|a+b| \leq |b|$. Then $a \oplus b = a+b$. (An analogous result holds for subtraction.)*

Proof: Without loss of generality, assume $|a| \geq |b|$. Clearly, the least significant nonzero bit of $a+b$ is no smaller in magnitude than $\text{ulp}(b)$. However, $|a+b| \leq |b|$. It follows that $a+b$ can be expressed in p bits. ∎

Many of the algorithms will rely on the following lemma, which shows that subtraction is exact for two operands within a factor of two of each other:

Lemma 5 (Sterbenz [26]) *Suppose that $b \in [\frac{a}{2}, 2a]$. Then $a \ominus b = a - b$.*

Proof: Without loss of generality, assume $|a| \geq |b|$. (The other case is symmetric, because $a \ominus b = -b \ominus -a$.) Then $b \in [\frac{a}{2}, a]$. The difference satisfies $|a-b| \leq |b| \leq |a|$; the result follows by Lemma 4. ∎

Two examples demonstrating Lemma 5 appear in Figure 2. If a and b have the same exponent, then floating-point subtraction is analogous to finding the difference between two p-bit integers of the same sign, and the result is expressible in p bits. Otherwise, the exponents of a and b differ by one, because $b \in [\frac{a}{2}, 2a]$. In this case, the difference has the smaller of the two exponents, and so can be expressed in p bits.

2.3 Simple 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, due to Dekker and Knuth respectively, are presented.

Theorem 6 (Dekker [7]) *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.*

$$
\begin{array}{rclcccccccccl}
a & = & & & & 1 & 1 & 1 & 1 & & & & \times 2^2 \\
b & = & & & & & & & 1 & 0 & 0 & 1 & \\
\cline{5-11}
x & = & a \oplus b & = & 1 & 0 & 0 & 1 & & & & & \times 2^3 \\
a & = & & & & 1 & 1 & 1 & 1 & & & & \times 2^2 \\
\cline{5-11}
b_{\text{virtual}} & = & x \ominus a & = & & & & 1 & 1 & 0 & 0 & & \\
 & & & & & & & & & & & & \\
y & = & b \ominus b_{\text{virtual}} & = & & & & & - & 1 & 1 & &
\end{array}
$$

Figure 3: Demonstration of FAST-TWO-SUM where a and b have the same sign. The sum of 111100 and 1001 is the expansion $1001000 + -11$.

$$
\begin{array}{rclcccccccl}
a & = & & & 1 & 0 & 0 & 1 & & \times 2^1 \\
b & = & & & - & 1 & 0 & 1 & 1 & \\
\cline{5-9}
x & = & a \oplus b & = & & & 1 & 1 & 1 & \\
a & = & & & 1 & 0 & 0 & 1 & & \times 2^1 \\
\cline{5-9}
b_{\text{virtual}} & = & x \ominus a & = & - & 1 & 0 & 1 & 1 & \\
 & & & & & & & & & \\
y & = & b \ominus b_{\text{virtual}} & = & & & & & 0 &
\end{array}
$$

Figure 4: Demonstration of FAST-TWO-SUM where a and b have opposite sign and $|b| > \frac{|a|}{2}$.

FAST-TWO-SUM(a, b)
1 $\quad x \Leftarrow a \oplus b$
2 $\quad b_{\text{virtual}} \Leftarrow x \ominus a$
3 $\quad y \Leftarrow b \ominus b_{\text{virtual}}$
4 $\quad$ **return** (x, y)

Proof: Line 1 computes $a + b$, but may be subject to rounding, so we have $x = a + b + \text{err}(a \oplus b)$. By assumption $|a| \geq |b|$, so a and x must have the same sign (or $x = 0$).

Line 2 computes the quantity b_{virtual}, which is the value that was *really* added to a in Line 1. This subtraction is computed exactly; this fact can be proven by considering two cases. If a and b have the same sign, or if $|b| \leq \frac{|a|}{2}$, then $x \in [\frac{a}{2}, 2a]$ and one can apply Lemma 5 (see Figure 3). On the other hand, if a and b are opposite in sign and $|b| > \frac{|a|}{2}$, then $b \in [-\frac{a}{2}, -a]$ and one can apply Lemma 5 to Line 1, showing that x was computed exactly and therefore $b_{\text{virtual}} = b$ (see Figure 4). In either case the subtraction is exact, so $b_{\text{virtual}} = x - a = b + \text{err}(a \oplus b)$.

Line 3 is also computed exactly. By Corollary 2, $b - b_{\text{virtual}} = -\text{err}(a \oplus b)$ is expressible in p bits.

It follows that $y = -\text{err}(a \oplus b)$ and $x = a + b + \text{err}(a \oplus b)$, hence $a + b = x + y$. Exact rounding guarantees that $|y| \leq \frac{1}{2}\text{ulp}(x)$, so x and y are nonoverlapping. ∎

Note that the outputs x and y do *not* necessarily have the same sign, as Figure 3 demonstrates. 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 proof of the correctness of this sequence is analogous to Theorem 6.

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. With most C compilers[3], perhaps the fastest portable way to implement this test is with the statement "`if ((a > b) == (a > -b))`". This test takes time to execute, and the slowdown may be surprisingly large because on modern pipelined and superscalar architectures, an **if** statement coupled with imperfect microprocessor branch prediction may cause a processor's instruction pipeline to drain. This explanation is speculative and machine-dependent, but the TWO-SUM algorithm below, which avoids a comparison at the cost of three additional floating-point operations, is usually empirically faster[4]. Of course, FAST-TWO-SUM remains faster if the relative sizes of the operands are known *a priori*, and the comparison can be avoided.

Theorem 7 (Knuth [17]) *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 $a+b = x+y$, where x is an approximation to $a+b$ and y is the roundoff error in the calculation of x.*

TWO-SUM(a, b)
1 $x \Leftarrow a \oplus b$
2 $b_{\text{virtual}} \Leftarrow x \ominus a$
3 $a_{\text{virtual}} \Leftarrow x \ominus b_{\text{virtual}}$
4 $b_{\text{roundoff}} \Leftarrow b \ominus b_{\text{virtual}}$
5 $a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$
6 $y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$
7 **return** (x, y)

Proof: If $|a| \geq |b|$, then Lines 1, 2, and 4 correspond precisely to the FAST-TWO-SUM algorithm. Recall from the proof of Theorem 6 that Line 2 is calculated exactly; it follows that Line 3 of TWO-SUM is calculated exactly as well, because $a_{\text{virtual}} = a$ can be expressed exactly. Hence, a_{roundoff} is zero, $y = b_{\text{roundoff}}$ is computed exactly, and the procedure is correct.

Now, suppose that $|a| < |b|$, and consider two cases. If $|x| < |a| < |b|$, then x is computed exactly by Lemma 4. It immediately follows that $b_{\text{virtual}} = b$, $a_{\text{virtual}} = a$, and b_{roundoff}, a_{roundoff}, and y are zero.

Conversely, if $|x| \geq |a|$, Lines 1 and 2 may be subject to rounding, so $x = a + b + \text{err}(a \oplus b)$, and $b_{\text{virtual}} = b + \text{err}(a \oplus b) + \text{err}(x \ominus a)$. (See Figure 5.) Lines 2, 3, and 5 are analogous to the three lines of FAST-TWO-DIFF (with Line 5 negated), so Lines 3 and 5 are computed exactly. Hence, $a_{\text{virtual}} = x - b_{\text{virtual}} = a - \text{err}(x \ominus a)$, and $a_{\text{roundoff}} = \text{err}(x \ominus a)$.

Because $|b| > |a|$, we have $|x| = |a \oplus b| \leq 2|b|$, so the roundoff errors $\text{err}(a \oplus b)$ and $\text{err}(x \ominus a)$ each cannot be more than $\text{ulp}(b)$, so $b_{\text{virtual}} \in [\frac{b}{2}, 2b]$ (for $p \geq 3$) and Lemma 5 can be applied to show that Line 4 is exact. Hence, $b_{\text{roundoff}} = -\text{err}(a \oplus b) - \text{err}(x \ominus a)$. Finally, Line 6 is exact because by Corollary 2, $a_{\text{roundoff}} + b_{\text{roundoff}} = -\text{err}(a \oplus b)$ is expressible in p bits.

It follows that $y = -\text{err}(a \oplus b)$ and $x = a + b + \text{err}(a \oplus b)$, hence $a + b = x + y$. ■

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

---

[3] The exceptions are those few that can identify and optimize the `fabs()` math library call.

[4] On a DEC Alpha-based workstation, using the bundled C compiler with optimization level 3, TWO-SUM uses roughly 65% as much time as FAST-TWO-SUM conditioned with the test "`if ((a > b) == (a > -b))`". On a SPARCstation IPX, using the GNU compiler with optimization level 2, TWO-SUM uses roughly 85% as much time. On the other hand, using the SPARCstation's bundled compiler with optimization (which produces slower code than gcc), conditional FAST-TWO-SUM uses only 82% as much time as TWO-SUM. The lesson is that for optimal speed, one must time each method with one's own machine and compiler.

$$
\begin{array}{rcccrrrrrrrrl}
a & = & & & & & & 1 & 1. & 1 & 1 & \\
b & = & & & & 1 & 1 & 0 & 1 & & & \\
\cline{5-11}
x & = & a \oplus b & = & 1 & 0 & 0 & 0 & & & & \times 2^1 \\
a & = & & & & & & 1 & 1. & 1 & 1 & \\
\cline{5-11}
b_{\text{virtual}} & = & x \ominus a & = & & 1 & 1 & 0 & 0 & & & \\
 & & & & & & & & & & & \\
a_{\text{virtual}} & = & x \ominus b_{\text{virtual}} & = & & & 1 & 0 & 0 & & & \\
 & & & & & & & & & & & \\
b_{\text{roundoff}} & = & b \ominus b_{\text{virtual}} & = & & & & & 1 & & & \\
a_{\text{roundoff}} & = & a \ominus a_{\text{virtual}} & = & & & & - & 0. & 0 & 1 & \\
\cline{5-11}
y & = & a_{\text{roundoff}} \oplus b_{\text{roundoff}} & = & & & & & 0. & 1 & 1 &
\end{array}
$$

Figure 5: Demonstration of TWO-SUM where $|a| < |b|$ and $|a| \leq |x|$. The sum of 11.11 and 1101 is the expansion $10000 + 0.11$.

Corollary 8 *Let x and y be the values returned by* FAST-TWO-SUM *or* TWO-SUM.

(a) If $|y| \geq 2^i$ for some integer i, then $|x + y| \geq 2^i(2^p + 1)$.

(b) If $|y| > 2^i$ for some integer i, then $|x + y| > 2^i(2^{p+1} + 1)$.

Proof: y is the roundoff error $-\text{err}(a \oplus b)$ for some a and b. By Theorems 6 and 7, $a + b = x + y$. The results follow directly from Lemma 3. ■

Corollary 9 *Let x and y be the values returned by* FAST-TWO-SUM *or* TWO-SUM. *On a machine whose arithmetic uses round-to-even tiebreaking, x and y are nonadjacent.*

Proof: Exact rounding guarantees that $y \leq \frac{1}{2}\text{ulp}(x)$. If the inequality is strict, x and y are nonadjacent. If $y = \frac{1}{2}\text{ulp}(x)$, the round-to-even rule ensures that the least significant bit of the significand of x is zero, so x and y are nonadjacent. ■

2.4 Expansion Addition

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. Three methods are available. EXPANSION-SUM adds an m-component expansion to an n-component expansion in $\mathcal{O}(mn)$ time. LINEAR-EXPANSION-SUM and FAST-EXPANSION-SUM do the same in $\mathcal{O}(m + n)$ time.

Despite its asymptotic disadvantage, EXPANSION-SUM can be faster than the linear-time algorithms in cases where the size of each expansion is small and fixed, because program loops can be completely unrolled and indirection overhead can be eliminated (by avoiding the use of arrays). The linear-time algorithms have conditionals that make such optimizations untenable. Hence, EXPANSION-SUM and FAST-EXPANSION-SUM are both used in the implementations of geometric predicates described in Section 4.

Expansion-Sum and Linear-Expansion-Sum both have the property that their outputs are nonoverlapping if their inputs are nonoverlapping, and nonadjacent if their inputs are nonadjacent. Fast-Expansion-Sum is faster than Linear-Expansion-Sum, performing six floating-point operations per component rather than nine, but has three disadvantages. First, Fast-Expansion-Sum does not always preserve either the nonoverlapping nor the nonadjacent property; instead, it preserves an intermediate property, described later. Second, whereas Linear-Expansion-Sum makes no assumption about the tiebreaking rule, Fast-Expansion-Sum is designed for machines that use round-to-even tiebreaking, and can fail on machines with other tiebreaking rules. Third, the correctness proof for Fast-Expansion-Sum is much more tedious. Nevertheless, I use Fast-Expansion-Sum in my geometric predicates, and relegate the slower Linear-Expansion-Sum to Appendix A. Users of machines that have exact rounding but not round-to-even tiebreaking should replace calls to Fast-Expansion-Sum with calls to Linear-Expansion-Sum.

A complicating characteristic of all the algorithms for manipulating expansions is that there may be spurious zero components scattered throughout the output expansions, even if no zeros were present in the input expansions. For instance, if the expansions $1111+0.0101$ and $1100+0.11$ are passed as inputs to any of the three expansion addition algorithms, the output expansion in four-bit arithmetic is $11100+0+0+0.0001$. One may want to add expansions thus produced to other expansions; fortunately, all the algorithms in this article cope well with spurious zero components in their input expansions. Unfortunately, accounting for these zero components could complicate the correctness proofs significantly. To avoid confusion, most of the proofs for the addition and scaling algorithms are written as if all input components are nonzero. Spurious zeros can be integrated into the proofs (after the fact) by noting that the effect of a zero input component is always to produce a zero output component without changing the value of the accumulator (denoted by the variable Q). The effect can be likened to a pipeline delay; it will become clear in the first few proofs.

Each algorithm has an accompanying dataflow diagram, like Figure 6. Readers will find the proofs easier to understand if they follow the diagrams while reading the proofs, and keep several facts in mind. First, Lemma 1 indicates that the down arrow from any Two-Sum box represents a number no larger than either input to the box. (This is why a zero input component yields a zero output component.) Second, Theorems 6 and 7 indicate that the down arrow from any Two-Sum box represents a number too small to overlap the number represented by the left arrow from the box.

I begin with an algorithm for adding a single p-bit value to an expansion.

Theorem 10 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, and let b be a p-bit value where $p \geq 3$. Suppose that the components $e_1, e_2, \ldots, e_m$ are sorted in order of* **increasing** *magnitude, except that any of the e_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{m+1} h_i = e + b$, where the components $h_1, h_2, \ldots, h_{m+1}$ are also in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e is nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

```
GROW-EXPANSION(e, b)
    Q_0 ⇐ b
    for i ⇐ 1 to m
        (Q_i, h_i) ⇐ TWO-SUM(Q_{i−1}, e_i)
    h_{m+1} ⇐ Q_m
    return h
```

Q_i is an approximate sum of b and the first i components of e; see Figure 6. In an implementation, the array Q can be collapsed into a single scalar.

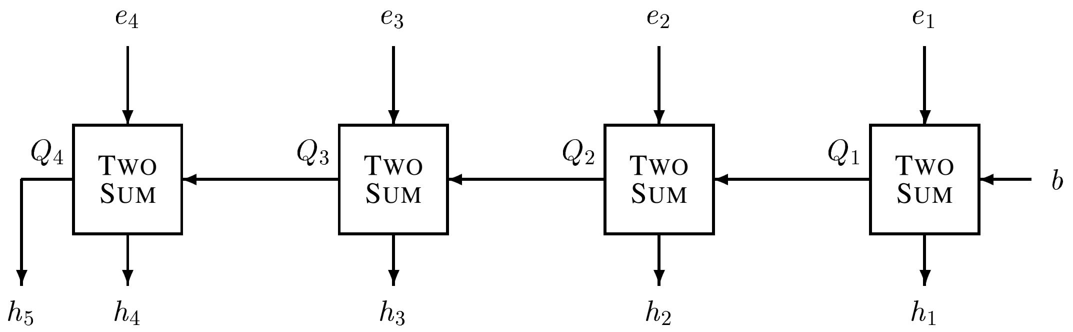

Figure 6: Operation of GROW-EXPANSION. The expansions e and h are illustrated with their most significant components on the left. All TWO-SUM boxes in this article observe the convention that the larger output (x) emerges from the left side of each box, and the smaller output (y) from the bottom or right. Each Q_i term is an approximate running total.

Proof: At the end of each iteration of the **for** loop, the invariant $Q_i + \sum_{j=1}^{i} h_j = b + \sum_{j=1}^{i} e_j$ holds. Certainly this invariant holds for $i = 0$ after Line 1 is executed. From Line 3 and Theorem 7, we have that $Q_i + h_i = Q_{i-1} + e_i$; from this one can deduce inductively that the invariant holds for all (relevant values of) i. Thus, after Line 4 is executed, $\sum_{j=1}^{m+1} h_j = \sum_{j=1}^{m} e_j + b$.

For all i, the output of TWO-SUM (in Line 3) has the property that h_i and Q_i do not overlap. By Lemma 1, $|h_i| \leq |e_i|$, and because e is a nonoverlapping expansion whose nonzero components are arranged in increasing order, h_i cannot overlap any of $e_{i+1}, e_{i+2}, \ldots$. It follows that h_i cannot overlap any of the later components of h, because these are constructed by summing Q_i with later e components. Hence, h is nonoverlapping and increasing (excepting zero components of h). If round-to-even tiebreaking is used, then h_i and Q_i are nonadjacent for all i (by Corollary 9), so if e is nonadjacent, then h is nonadjacent.

If any of the e_i is zero, the corresponding output component h_i is also zero, and the accumulator value Q is unchanged ($Q_i = Q_{i-1}$). (For instance, consider Figure 6, and suppose that e_3 is zero. The accumulator value Q_2 shifts through the pipeline to become Q_3, and a zero is harmlessly output as h_3. The same effect occurs in several algorithms in this article.) ■

Corollary 11 *The first m components of h are each no larger than the corresponding component of e. (That is, $|h_1| \leq |e_1|, |h_2| \leq |e_2|, \ldots, |h_m| \leq |e_m|$.) Furthermore, $|h_1| \leq |b|$.*

Proof: Follows immediately by application of Lemma 1 to Line 3. (Both of these facts are apparent in Figure 6. Recall that the down arrow from any TWO-SUM box represents a number no larger than either input to the box.) ■

If e is a long expansion, two optimizations might be advantageous. The first is to use a binary search to find the smallest component of e greater than or equal to $\text{ulp}(b)$, and start there. A variant of this idea, without the search, is used in the next theorem. The second optimization is to stop early if the output of a TWO-SUM operation is the same as its inputs; the expansion is already nonoverlapping.

A naïve way to add one expansion to another is to repeatedly use GROW-EXPANSION to add each component of one expansion to the other. One can improve this idea with a small modification.

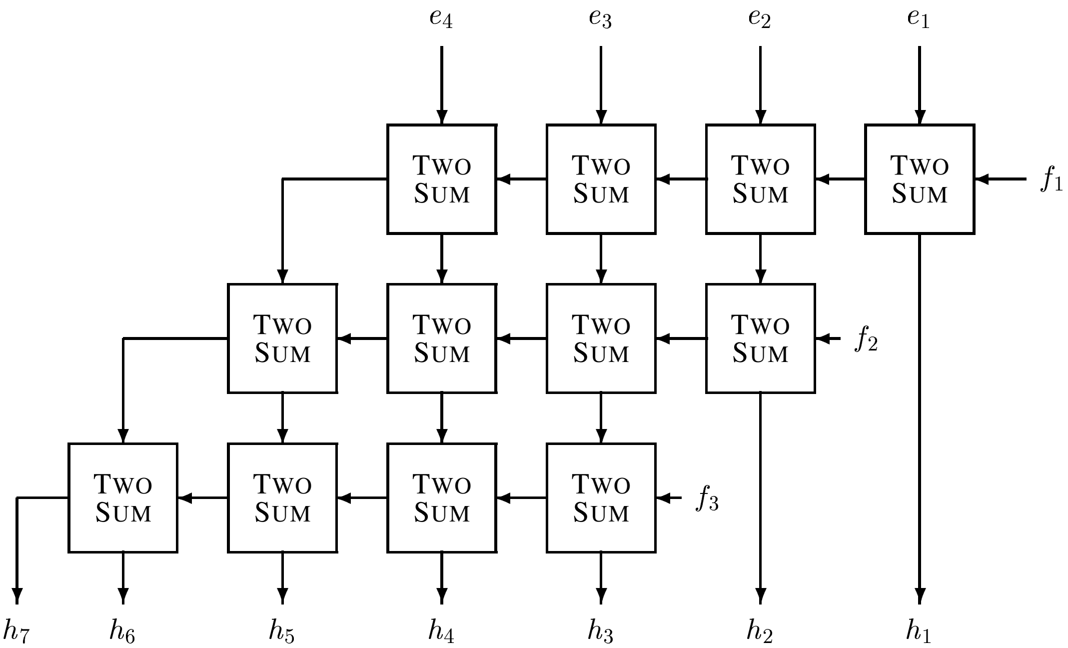

Figure 7: Operation of EXPANSION-SUM.

Theorem 12 *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 $p \geq 3$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. Then 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 in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e and f are nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

EXPANSION-SUM(e, f)
1 $\quad h \Leftarrow e$
2 $\quad$ **for** $i \Leftarrow 1$ **to** n
3 $\quad\quad \langle h_i, h_{i+1}, \ldots, h_{i+m} \rangle \Leftarrow$ GROW-EXPANSION$(\langle h_i, h_{i+1}, \ldots, h_{i+m-1} \rangle, f_i)$
4 $\quad$ **return** h

Proof: That $\sum_{i=1}^{m+n} h_i = \sum_{i=1}^{m} e_i + \sum_{i=1}^{n} f_i$ upon completion can be proven by induction on Line 3.

After setting $h \Leftarrow e$, EXPANSION-SUM traverses the expansion f from smallest to largest component, individually adding these components to h using GROW-EXPANSION (see Figure 7). The theorem would follow directly from Theorem 10 if each component f_i were added to the whole expansion h, but to save time, only the subexpansion $\langle h_i, h_{i+1}, \ldots, h_{i+m-1} \rangle$ is considered. (In Figure 7, this optimization saves three TWO-SUM operations that would otherwise appear in the lower right corner of the figure.)

When f_i is considered, the components $f_1, f_2, \ldots, f_{i-1}$ have already been summed into h. According to Corollary 11, $|h_j| \leq |f_j|$ after iteration j of Line 3. Because f is an increasing nonoverlapping expansion, for any $j < i$, h_j cannot overlap f_i, and furthermore $|h_j| < |f_i|$ (unless $f_i = 0$). Therefore, when one sums f_i into h, one can skip the first $i - 1$ components of h without sacrificing the nonoverlapping and increasing properties of h. Similarly, if e and f are each nonadjacent, one can skip the first $i - 1$ components of h without sacrificing the nonadjacent property of h.

No difficulty ensues if f_i is a spurious zero component, because zero does not overlap any number. GROW-EXPANSION will deposit a zero at h_i and continue normally. ∎

Unlike EXPANSION-SUM, FAST-EXPANSION-SUM does not preserve the nonoverlapping or nonadjacent properties, but it is guaranteed to produce a strongly nonoverlapping output if its inputs are strongly nonoverlapping. An expansion is *strongly nonoverlapping* if no two of its components are overlapping, no component is adjacent to two other components, and any pair of adjacent components have the property that both components can be expressed with a one-bit significand (that is, both are powers of two). For instance, $11000 + 11$ and $10000 + 1000 + 10 + 1$ are both strongly nonoverlapping, but $11100 + 11$ is not, nor is $100 + 10 + 1$. A characteristic of this property is that a zero bit must occur in the expansion at least once every $p + 1$ bits. For instance, in four-bit arithmetic, a strongly nonoverlapping expansion whose largest component is 1111 can be no greater than $1111.01111011110\ldots$. Any nonadjacent expansion is strongly nonoverlapping, and any strongly nonoverlapping expansion is nonoverlapping, but the converse implications do not apply. Recall that any floating-point value has a nonadjacent expansion; hence, any floating-point value has a strongly nonoverlapping expansion. For example, 1111.1 may be expressed as $10000 + -0.1$.

Under the assumption that all expansions are strongly nonoverlapping, it is possible to prove the first key result of this article: the FAST-EXPANSION-SUM algorithm defined below behaves correctly under round-to-even tiebreaking. The algorithm can also be used with round-toward-zero arithmetic, but the proof is different. I have emphasized round-to-even arithmetic here due to the IEEE 754 standard.

A variant of this algorithm was presented by Priest [23], but it is used differently here. Priest uses the algorithm to sum two nonoverlapping expansions, and proves under general conditions that the components of the resulting 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. Here, by contrast, the algorithm is used to sum two strongly nonoverlapping expansions, and the result is also a strongly nonoverlapping expansion. Not surprisingly, the proof demands more stringent conditions than Priest requires: binary arithmetic with exact rounding and round-to-even tiebreaking, consonant with the IEEE 754 standard. No renormalization is needed.

Theorem 13 *Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be strongly nonoverlapping expansions of m and n p-bit components, respectively, where $p \geq 4$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. On a machine whose arithmetic uses the round-to-even rule, the following algorithm will produce a strongly 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.*

FAST-EXPANSION-SUM(e, f)
Merge e and f into a single sequence g, in order of
  nondecreasing magnitude (possibly with interspersed zeros)
$(Q_2, h_1) \Leftarrow$ FAST-TWO-SUM(g_2, g_1)
for $i \Leftarrow 3$ **to** $m + n$
  $(Q_i, h_{i-1}) \Leftarrow$ TWO-SUM(Q_{i-1}, g_i)
$h_{m+n} \Leftarrow Q_{m+n}$
return h

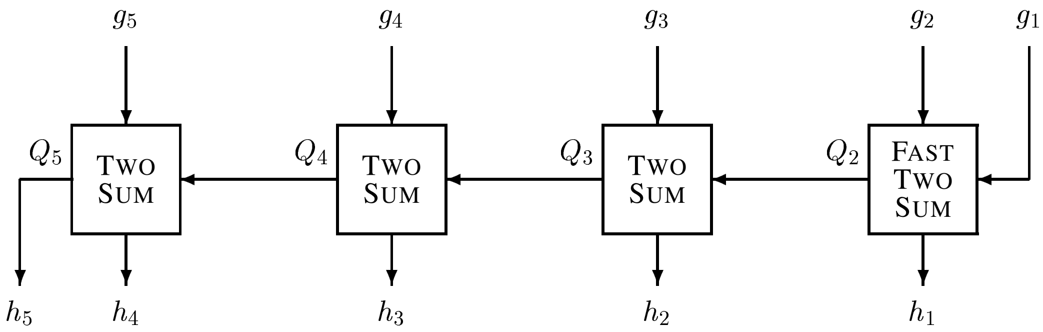

Figure 8: Operation of FAST-EXPANSION-SUM. The Q_i terms maintain an approximate running total.

Q_i is an approximate sum of the first i components of g; see Figure 8.

Several lemmata will aid the proof of Theorem 13. I begin with a proof that the sum itself is correct.

Lemma 14 (Q Invariant) *At the end of each iteration of the* **for** *loop, the invariant* $Q_i + \sum_{j=1}^{i-1} h_j = \sum_{j=1}^{i} g_j$ *holds. This assures us that after Line 5 is executed,* $\sum_{j=1}^{m+n} h_j = \sum_{j=1}^{m+n} g_j$*, so the algorithm produces a correct sum.*

Proof: The invariant clearly holds for $i = 2$ after Line 2 is executed. For larger values of i, Line 4 ensures that $Q_i + h_{i-1} = Q_{i-1} + g_i$; the invariant follows by induction. ∎

Lemma 15 *Let* $\widehat{g} = \sum_{j=1}^{k} \widehat{g}_j$ *be a series formed by merging two strongly nonoverlapping expansions, or a subseries thereof. Suppose that* $\widehat{g}_k$ *is the largest component and has a nonzero bit of magnitude* 2^i *or smaller for some integer* i*. Then* $|\sum_{j=1}^{k} \widehat{g}_j| < 2^i(2^{p+1} - 1)$*, and* $|\sum_{j=1}^{k-1} \widehat{g}_j| < 2^i(2^p)$.

Proof: Let $\widehat{e}$ and $\widehat{f}$ be the expansions (or subsequences thereof) from which $\widehat{g}$ was formed, and assume that the component $\widehat{g}_k$ comes from the expansion $\widehat{e}$. Because $\widehat{g}_k$ is the largest component of $\widehat{e}$ and has a nonzero bit of magnitude 2^i or smaller, and because $\widehat{e}$ is strongly nonoverlapping, $|\widehat{e}|$ is less than $2^i(2^p - \frac{1}{2})$. (For instance, if $p = 4$ and $i = 0$, then $|\widehat{e}| \leq 1111.0111101111\ldots$.) The same bound applies to the expansion $\widehat{f}$, so $|\widehat{g}| = |\widehat{e} + \widehat{f}| < 2^i(2^{p+1} - 1)$.

If we omit $\widehat{g}_k$ from the sum, there are two cases to consider. If $\widehat{g}_k = 2^i$, then $|\widehat{e} - \widehat{g}_k|$ is less than 2^i, and $|\widehat{f}|$ is less than $2^i(2)$. (For instance, if $p = 4$, $i = 0$, and $\widehat{g}_k = 1$, then $|\widehat{e} - \widehat{g}_k| \leq 0.10111101111\ldots$, and $|\widehat{f}| \leq 1.10111101111\ldots$.) Conversely, if $\widehat{g}_k \neq 2^i$, then $|\widehat{e} - \widehat{g}_k|$ is less than $2^i(\frac{1}{2})$, and $|\widehat{f}|$ is less than $2^i(2^p - \frac{1}{2})$. (For instance, if $p = 4$, $i = 0$, and $\widehat{g}_k = 1111$, then $|\widehat{e} - \widehat{g}_k| \leq 0.0111101111\ldots$, and $|\widehat{f}| \leq 1111.0111101111\ldots$.) In either case, $|\widehat{g} - \widehat{g}_k| = |\widehat{e} - \widehat{g}_k + \widehat{f}| < 2^i(2^p)$. ∎

Lemma 16 *The expansion* h *produced by* FAST-EXPANSION-SUM *is a nonoverlapping expansion whose components are in order of increasing magnitude (excepting zeros).*

$$
\begin{array}{rr}
\left|\sum_{j=1}^{i} g_i\right| \left\{ \begin{array}{r} \left|\sum_{j'} e_{j'}\right| \leq \\ \left|\sum_{j''} f_{j''}\right| \leq \end{array} \right. & \begin{array}{r} \boxed{\quad g_{i+1} \quad}.\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \\ 1\ 1\ 1\ 1.\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \end{array} \\
\left|\sum_{j=1}^{i-2} h_j\right| \leq & 0.\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\
\hline
\left|Q_i + h_{i-1}\right| \leq & 1\ 0\ 0\ 0\ 0.\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1
\end{array}
$$

Figure 9: Demonstration (for $p = 4$) of how the Q Invariant is used in the proof that h is nonoverlapping. The top two values, e and f, are being summed to form h. Because g_{i+1} has a nonzero bit of magnitude no greater than 1, and because g is formed by merging two strongly nonoverlapping expansions, the sum $|\sum_{j=1}^{i} g_i| + |\sum_{j=1}^{i-2} h_j|$ can be no larger than illustrated in this worst-case example. As a result, $|Q_i + h_{i-1}|$ cannot be large enough to have a roundoff error of 1, so $|h_{i-1}|$ is smaller than 1 and cannot overlap g_{i+1}. (Note that g_{i+1} is not part of the sum; it appears above in a box drawn as a placeholder that bounds the value of each expansion.)

Proof: Suppose for the sake of contradiction that two successive nonzero components of h overlap or occur in order of decreasing magnitude. Denote the first such pair produced[5] h_{i-1} and h_i; then the components $h_1, \ldots, h_{i-1}$ are nonoverlapping and increasing (excepting zeros).

Assume without loss of generality that the exponent of h_{i-1} is zero, so that h_{i-1} is of the form $\pm 1.*$, where an asterisk represents a sequence of arbitrary bits.

Q_i and h_{i-1} are produced by a TWO-SUM or FAST-TWO-SUM operation, and are therefore nonadjacent by Corollary 9 (because the round-to-even rule is used). Q_i is therefore of the form $\pm * 00$ (having no bits of magnitude smaller than four). Because $|h_{i-1}| \geq 1$, Corollary 8(a) guarantees that

$$|Q_i + h_{i-1}| \geq 2^p + 1. \tag{1}$$

Because the offending components h_{i-1} and h_i are nonzero and either overlapping or of decreasing magnitude, there must be at least one nonzero bit in the significand of h_i whose magnitude is no greater than one. One may ask, where does this offending bit come from? h_i is computed by Line 4 from Q_i and g_{i+1}, and the offending bit cannot come from Q_i (which is of the form $\pm * 00$), so it must have come from g_{i+1}. Hence, $|g_{i+1}|$ has a nonzero bit of magnitude one or smaller. Applying Lemma 15, one finds that $|\sum_{j=1}^{i} g_j| < 2^p$.

A bound for $\sum_{j=1}^{i-2} h_j$ can be derived by recalling that h_{i-1} is of the form $\pm 1.*$, and $h_1, \ldots, h_{i-1}$ are nonoverlapping and increasing. Hence, $|\sum_{j=1}^{i-2} h_j| < 1$.

Rewrite the Q Invariant in the form $Q_i + h_{i-1} = \sum_{j=1}^{i} g_j - \sum_{j=1}^{i-2} h_j$. Using the bounds derived above, we obtain

$$|Q_i + h_{i-1}| < 2^p + 1. \tag{2}$$

See Figure 9 for a concrete example.

Inequalities 1 and 2 cannot hold simultaneously. The result follows by contradiction. ■

[5] It is implicitly assumed here that the first offending pair is not separated by intervening zeros. The proof could be written to consider the case where intervening zeros appear, but this would make it even more convoluted. Trust me.

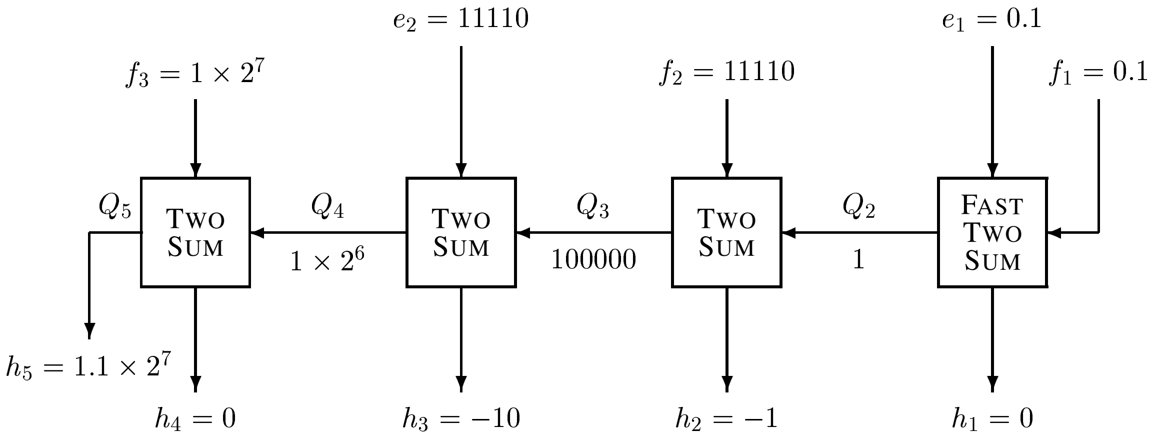

Figure 10: A four-bit example where FAST-EXPANSION-SUM generates two adjacent components h_2 and h_3. The figure permits me a stab at explaining the (admittedly thin) intuition behind Theorem 13: suppose h_2 is of the form $\pm 1.*$. Because h_2 is the roundoff term associated with Q_3, Q_3 must be of the form $*00$ if round-to-even arithmetic is used. Hence, the bit of magnitude 2 in h_3 must have come from e_2. This implies that $|e_2|$ is no larger than 11110, which imposes bounds on how large $|Q_3|$ and $|Q_4|$ can be (Lemma 15); these bounds in turn imply that $|h_2|$ can be no larger than 1, and $|h_3|$ can be no larger than 10. Furthermore, h_4 cannot be adjacent to h_3 because neither Q_4 nor f_3 can have a bit of magnitude 4.

Proof of Theorem 13: Lemma 14 ensures that $h = e + f$. Lemma 16 eliminates the possibility that the components of h overlap or fail to occur in order of increasing magnitude; it remains only to prove that h is strongly nonoverlapping. Suppose that two successive nonzero components h_{i-1} and h_i are adjacent.

Assume without loss of generality that the exponent of h_{i-1} is zero, so that h_{i-1} is of the form $\pm 1.*$. As in the proof of Lemma 16, Q_i must have the form $\pm * 00$.

Because h_{i-1} and h_i are adjacent, the least significant nonzero bit of h_i has magnitude two; that is, h_i is of the form $\pm * 10$. Again we ask, where does this bit come from? As before, this bit cannot come from Q_i, so it must have come from g_{i+1}. Hence, $|g_{i+1}|$ has a nonzero bit of magnitude two. Applying Lemma 15, we find that $|\sum_{j=1}^{i+1} g_j| < 2^{p+2} - 2$ and $|\sum_{j=1}^{i} g_j| < 2^{p+1}$.

Bounds for $\sum_{j=1}^{i-1} h_j$ and $\sum_{j=1}^{i-2} h_j$ can also be derived by recalling that h_{i-1} is of the form $\pm 1.*$ and is the largest component of a nonoverlapping expansion. Hence, $|\sum_{j=1}^{i-1} h_j| < 2$, and $|\sum_{j=1}^{i-2} h_j| < 1$.

Rewriting the Q Invariant in the form $Q_{i+1} + h_i = \sum_{j=1}^{i+1} g_j - \sum_{j=1}^{i-1} h_j$, we obtain

$$|Q_{i+1} + h_i| < 2^{p+2}. \tag{3}$$

The Q Invariant also gives us the identity $Q_i + h_{i-1} = \sum_{j=1}^{i} g_j - \sum_{j=1}^{i-2} h_j$. Hence,

$$|Q_i + h_{i-1}| < 2^{p+1} + 1. \tag{4}$$

Recall that the value $|h_i|$ is at least 2. Consider the possibility that $|h_i|$ might be greater than 2; by Corollary 8(b), this can occur only if $|Q_{i+1} + h_i| > 2^{p+2} + 2$, contradicting Inequality 3. Hence, $|h_i|$ must be exactly 2, and is expressible in one bit. (Figure 10 gives an example where this occurs.)

Similarly, the value $|h_{i-1}|$ is at least 1. Consider the possibility that $|h_{i-1}|$ might be greater than 1; by Corollary 8(b), this can occur only if $|Q_i + h_{i-1}| > 2^{p+1} + 1$, contradicting Inequality 4. Hence, $|h_{i-1}|$ must be exactly 1, and is expressible in one bit.

By Corollary 8(a), $|Q_i + h_{i-1}| \geq 2^p + 1$ (because $|h_{i-1}| = 1$). Using this inequality, the inequality $|\sum_{j=1}^{i-2} h_j| < 1$, and the Q Invariant, one can deduce that $|\sum_{j=1}^{i} g_j| > 2^p$. Because g is formed from two nonoverlapping increasing expansions, this inequality implies that $|g_i| \geq 2^{p-2} \geq 100$ binary (recalling that $p \geq 4$), and hence $g_{i+2}, g_{i+3}, \ldots$ must all be of the form $\pm * 000$ (having no bits of magnitude smaller than 8). Q_{i+1} is also of the form $\pm * 000$, because Q_{i+1} and h_i are produced by a TWO-SUM or FAST-TWO-SUM operation, and are therefore nonadjacent by Corollary 9 (assuming the round-to-even rule is used).

Because Q_{i+1} and $g_{i+2}, g_{i+3}, \ldots$ are of the form $\pm * 000$, $h_{i+1}, h_{i+2}, \ldots$ must be as well, and are therefore not adjacent to h_i. It follows that h cannot contain three consecutive adjacent components.

These arguments prove that if two components of h are adjacent, both are expressible in one bit, and no other components are adjacent to them. Hence, h is strongly nonoverlapping. ■

The proof of Theorem 13 is more complex than one would like. It is unfortunate that the proof requires strongly nonoverlapping expansions; it would be more parsimonious if FAST-EXPANSION-SUM produced nonoverlapping output from nonoverlapping input, or nonadjacent output from nonadjacent input. Unfortunately, it does neither. For a counterexample to the former possibility, consider adding the nonoverlapping expansion $11110000 + 1111 + 0.1111$ to itself in four-bit arithmetic. (This example produces an overlapping expansion if one uses the round-to-even rule, but not if one uses the round-toward-zero rule.) For a counterexample to the latter possibility, see Figure 10. On a personal note, it took me quite a bit of effort to find a property between nonoverlapping and nonadjacent that is preserved by FAST-EXPANSION-SUM. Several conjectures were laboriously examined and discarded before I converged on the strongly nonoverlapping property. I persisted only because the algorithm consistently works in practice.

It is also unfortunate that the proof requires explicit consideration of the tiebreaking rule. FAST-EXPANSION-SUM works just as well on a machine that uses the round-toward-zero rule. The conditions under which it works are also simpler—the output expansion is guaranteed to be nonoverlapping if the input expansions are. One might hope to prove that FAST-EXPANSION-SUM works regardless of rounding mode, but this is not possible. Appendix B demonstrates the difficulty with an example of how mixing round-toward-zero and round-to-even arithmetic can lead to the creation of overlapping expansions.

The algorithms EXPANSION-SUM and FAST-EXPANSION-SUM can be mixed only to a limited degree. EXPANSION-SUM preserves the nonoverlapping and nonadjacent properties, but not the strongly nonoverlapping property; FAST-EXPANSION-SUM preserves only the strongly nonoverlapping property. Because nonadjacent expansions are strongly nonoverlapping, and strongly nonoverlapping expansions are nonoverlapping, expansions produced exclusively by one of the two algorithms can be fed as input to the other, but it may be dangerous to repeatedly switch back and forth between the two algorithms. In practice, EXPANSION-SUM is only preferred for producing small expansions, which are nonadjacent and hence suitable as input to FAST-EXPANSION-SUM.

It is useful to consider the operation counts of the algorithms. EXPANSION-SUM uses mn TWO-SUM operations, for a total of $6mn$ flops (floating-point operations). FAST-EXPANSION-SUM uses $m + n - 2$ TWO-SUM operations and one FAST-TWO-SUM operation, for a total of $6m + 6n - 9$ flops. However, the merge step of FAST-EXPANSION-SUM requires $m+n-1$ comparison operations of the form "**if** $|e_i| > |f_j|$". Empirically, each such comparison seems to take roughly as long as three flops; hence, a rough measure is to estimate that FAST-EXPANSION-SUM takes as long to execute as $9m + 9n - 12$ flops.

These estimates correlate well with the measured performance of the algorithms. I implemented each procedure as a function call whose parameters are variable-length expansions stored as arrays, and measured them on a DEC Alpha-based workstation using the bundled compiler with optimization level 3. By plotting their performance over a variety of expansion sizes and fitting curves, I found that EXPANSION-SUM runs in $0.83(m+n) - 0.7$ microseconds, and FAST-EXPANSION-SUM runs in $0.54mn + 0.6$ microseconds. FAST-EXPANSION-SUM is always faster except when one of the expansions has only one component, in which case GROW-EXPANSION should be used.

As I have mentioned, however, the balance shifts when expansion lengths are small and fixed. By storing small, fixed-length expansions as scalar variables rather than arrays, one can unroll the loops in EXPANSION-SUM, remove array indexing overhead, and allow components to be allocated to registers by the compiler. Thus, EXPANSION-SUM is attractive in this special case, and is used to advantage in my implementation of the geometric predicates of Section 4. Note that FAST-EXPANSION-SUM is difficult to unroll because of the conditionals in its initial merging step.

On the other hand, the use of arrays to store expansions (and non-unrolled loops to manage them) confers the advantage that spurious zero components can easily be eliminated from output expansions. In the procedures GROW-EXPANSION, EXPANSION-SUM, and FAST-EXPANSION-SUM, as well as the procedures SCALE-EXPANSION and COMPRESS in the sections to come, *zero elimination* can be achieved by maintaining a separate index for the output array h and advancing this index only when the procedure produces a nonzero component of h. In practice, versions of these algorithms that eliminate zeros are almost always preferable to versions that don't (except when loop unrolling confers a greater advantage). Zero elimination adds a small amount of overhead for testing and indexing, but the lost time is virtually always regained when further operations are performed on the resulting shortened expansions.

Experience suggests that it is economical to use unrolled versions of EXPANSION-SUM to form expansions of up to about four components, tolerating interspersed zeros, and to use FAST-EXPANSION-SUM with zero elimination when forming (potentially) larger expansions.

2.5 Simple 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. The following theorem was first proven by Dekker [7]:

Theorem 17 *Let a be a p-bit floating-point number, where $p \geq 3$. Choose a* splitting point *s such that $\frac{p}{2} \leq s \leq p-1$. Then the following algorithm will produce a $(p-s)$-bit value a_{hi} and a nonoverlapping $(s-1)$-bit value a_{lo} such that $|a_{\text{hi}}| \geq |a_{\text{lo}}|$ and $a = a_{\text{hi}} + a_{\text{lo}}$.*

SPLIT(a, s)
$c \Leftarrow (2^s + 1) \otimes a$
$a_{\text{big}} \Leftarrow c \ominus a$
$a_{\text{hi}} \Leftarrow c \ominus a_{\text{big}}$
$a_{\text{lo}} \Leftarrow a \ominus a_{\text{hi}}$
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

$$
\begin{array}{rcccccccccccccl}
a & = & & & & & & & 1 & 1 & 1 & 0 & 1 & \\
2^3 a & = & & & & & 1 & 1 & 1 & 0 & 1 & & & & \times 2^3 \\
\cline{5-13}
c & = & (2^3+1)\otimes a & = & 1 & 0 & 0 & 0 & 0 & & & & & & \times 2^4 \\
a & = & & & & & & & 1 & 1 & 1 & 0 & 1 & \\
\cline{5-13}
a_{\text{big}} & = & c \ominus a & = & & 1 & 1 & 1 & 0 & 0 & & & & & \times 2^3 \\
\\
a_{\text{hi}} & = & c \ominus a_{\text{big}} & = & & & & 1 & 0 & 0 & 0 & 0 & & & \times 2^1 \\
\\
a_{\text{lo}} & = & a \ominus a_{\text{hi}} & = & & & & & & & - & 1 & 1 &
\end{array}
$$

Figure 11: Demonstration of SPLIT splitting a five-bit number into two two-bit numbers.

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.

Proof: Line 1 is equivalent to computing $2^s a \oplus a$. (Clearly, $2^s a$ can be expressed exactly, because multiplying a value by a power of two only changes its exponent, and does not change its significand.) Line 1 is subject to rounding, so we have $c = 2^s a + a + \text{err}(2^s a \oplus a)$.

Line 2 is also subject to rounding, so $a_{\text{big}} = 2^s a + \text{err}(2^s a \oplus a) + \text{err}(c \ominus a)$. It will become apparent shortly that the proof relies on showing that the exponent of a_{big} is no greater than the exponent of $2^s a$. Both $|\text{err}(2^s a \oplus a)|$ and $|\text{err}(c \ominus a)|$ are bounded by $\frac{1}{2}\text{ulp}(c)$, so the exponent of a_{big} can only be larger than that of $2^s a$ if every bit of the significand of a is nonzero except possibly the last (in four-bit arithmetic, a must have significand 1110 or 1111). By manually checking the behavior of SPLIT in these two cases, one can verify that the exponent of a_{big} is never larger than that of $2^s a$.

The reason this fact is useful is because, with Line 2, it implies that $|\text{err}(c \ominus a)| \leq \frac{1}{2}\text{ulp}(2^s a)$, and so the error term $\text{err}(c \ominus a)$ is expressible in $s - 1$ bits (for $s \geq 2$).

By Lemma 5, Lines 3 and 4 are calculated exactly. It follows that $a_{\text{hi}} = a - \text{err}(c \ominus a)$, and $a_{\text{lo}} = \text{err}(c \ominus a)$; the latter is expressible in $s - 1$ bits. To show that a_{hi} is expressible in $p - s$ bits, consider that its least significant bit cannot be smaller than $\text{ulp}(a_{\text{big}}) = 2^s \text{ulp}(a)$. If a_{hi} has the same exponent as a, then a_{hi} must be expressible in $p - s$ bits; alternatively, if a_{hi} has an exponent one greater than that of a (because $a - \text{err}(c \ominus a)$ has a larger exponent than a), then a_{hi} is expressible in one bit (as demonstrated in Figure 11).

Finally, the exactness of Line 4 implies that $a = a_{\text{hi}} + a_{\text{lo}}$ as required. ∎

Multiplication is performed by setting $s = \lceil \frac{p}{2} \rceil$, so that the p-bit operands a and b are each split into two $\lfloor \frac{p}{2} \rfloor$-bit pieces, a_{hi}, a_{lo}, b_{hi}, and b_{lo}. 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. These could then be summed using the FAST-EXPANSION-SUM procedure in Section 2.4. However, Dekker [7] provides several faster ways to accomplish the computation. Dekker attributes the following method to G. W. Veltkamp.

Theorem 18 *Let a and b be p-bit floating-point numbers, where $p \geq 6$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$, where x is an approximation to ab and y represents the roundoff error in the calculation of x. Furthermore, if round-to-even tiebreaking is used, x and y are nonadjacent. (See Figure 12.)*

$$
\begin{array}{rclc|cccccccccccc|l}
a & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
b & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
x & = & a \otimes b & = & 1 & 1 & 0 & 1 & 1 & 0 & & & & & & & \times 2^6 \\
 & & a_{\text{hi}} \otimes b_{\text{hi}} & = & 1 & 1 & 0 & 0 & 0 & 1 & & & & & & & \times 2^6 \\
err_1 & = & x \ominus (a_{\text{hi}} \otimes b_{\text{hi}}) & = & & & & 1 & 0 & 1 & 0 & 0 & 0 & & & & \times 2^3 \\
 & & a_{\text{lo}} \otimes b_{\text{hi}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
err_2 & = & err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}}) & = & & & & & 1 & 0 & 0 & 1 & 1 & 0 & & & \times 2^2 \\
 & & a_{\text{hi}} \otimes b_{\text{lo}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
err_3 & = & err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 0 & 0 & 0 & 0 & \\
 & & a_{\text{lo}} \otimes b_{\text{lo}} & = & & & & & & & & & 1 & 0 & 0 & 1 & \\
-y & = & err_3 \ominus (a_{\text{lo}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 1 & 0 & 0 & 1 &
\end{array}
$$

Figure 12: Demonstration of TWO-PRODUCT in six-bit arithmetic where $a = b = 111011$, $a_{\text{hi}} = b_{\text{hi}} = 111000$, and $a_{\text{lo}} = b_{\text{lo}} = 11$. Note that each intermediate result is expressible in six bits. The resulting expansion is $110110 \times 2^6 + 11001$.

TWO-PRODUCT(a, b)

1 $x \Leftarrow a \otimes b$
2 $(a_{\text{hi}}, a_{\text{lo}}) = \text{SPLIT}(a, \lceil \frac{p}{2} \rceil)$
3 $(b_{\text{hi}}, b_{\text{lo}}) = \text{SPLIT}(b, \lceil \frac{p}{2} \rceil)$
4 $err_1 \Leftarrow x \ominus (a_{\text{hi}} \otimes b_{\text{hi}})$
5 $err_2 \Leftarrow err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}})$
6 $err_3 \Leftarrow err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}})$
7 $y \Leftarrow (a_{\text{lo}} \otimes b_{\text{lo}}) \ominus err_3$
8 **return** (x, y)

Proof: Line 1 is subject to rounding, so we have $x = ab + \text{err}(a \otimes b)$. The multiplications in Lines 4 through 7 are all exact, because each factor has no more than $\lfloor \frac{p}{2} \rfloor$ bits; it will be proven that each of the subtractions is also exact, and thus $y = -\text{err}(a \otimes b)$.

Without loss of generality, assume that the exponents of a and b are $p - 1$, so that $|a|$ and $|b|$ are integers in the range $[2^{p-1}, 2^p - 1]$. In the proof of Theorem 17 it emerged that $|a_{\text{hi}}|$ and $|b_{\text{hi}}|$ are integers in the range $[2^{p-1}, 2^p]$, and $|a_{\text{lo}}|$ and $|b_{\text{lo}}|$ are integers in the range $[0, 2^{\lceil p/2 \rceil - 1}]$. From these ranges and the assumption that $p \geq 6$, one can derive the inequalities $|a_{\text{lo}}| \leq \frac{1}{8}|a_{\text{hi}}|$, $|b_{\text{lo}}| \leq \frac{1}{8}|b_{\text{hi}}|$, and $\text{err}(a \otimes b) \leq 2^{p-1} \leq \frac{1}{32}|a_{\text{hi}} b_{\text{hi}}|$.

Intuitively, $a_{\text{hi}} b_{\text{hi}}$ ought to be within a factor of two of $a \otimes b$, so that Line 4 is computed exactly (by Lemma 5). To confirm this hunch, note that $x = ab + \text{err}(a \otimes b) = a_{\text{hi}} b_{\text{hi}} + a_{\text{lo}} b_{\text{hi}} + a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b) = a_{\text{hi}} b_{\text{hi}} \pm \frac{19}{64}|a_{\text{hi}} b_{\text{hi}}|$ (using the inequalities stated above), which justifies the use of Lemma 5. Because Line 4 is computed without roundoff, $err_1 = a_{\text{lo}} b_{\text{hi}} + a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$.

We are assured that Line 5 is executed without roundoff error if the value $err_1 - a_{\text{lo}} b_{\text{hi}} = a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$ is expressible in p bits. I prove that this property holds by showing that the left-hand expression is a multiple of $2^{\lceil p/2 \rceil}$, and the right-hand expression is strictly smaller than $2^{\lceil 3p/2 \rceil}$.

The upper bound on the absolute value of the right-hand expression follows immediately from the upper bounds for a_{hi}, a_{lo}, b_{lo}, and $\text{err}(a \otimes b)$. To show that the left-hand expression is a multiple of $2^{\lceil p/2 \rceil}$, consider that err_1 must be a multiple of 2^{p-1} because $a \otimes b$ and $a_{\text{hi}} b_{\text{hi}}$ have exponents of at least $2p - 2$.

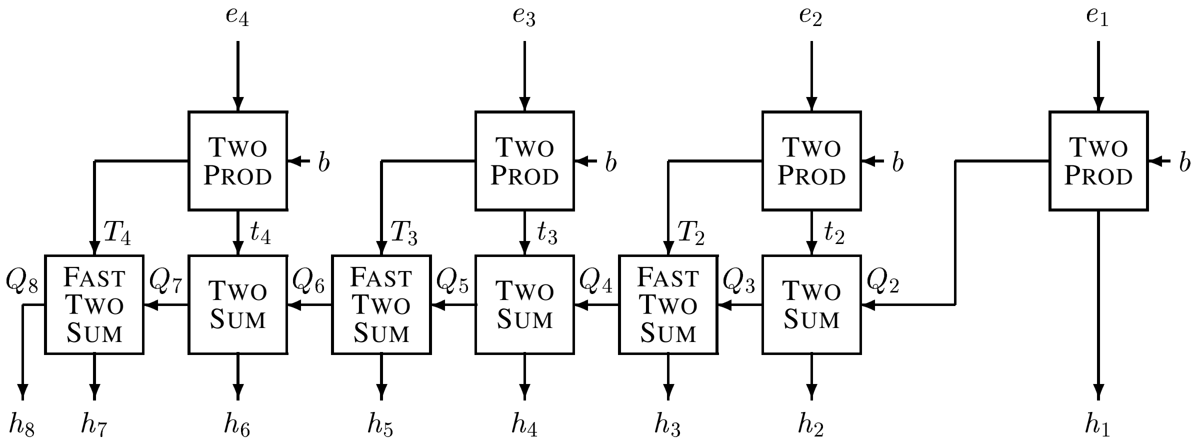

Figure 13: Operation of SCALE-EXPANSION.

Hence, $err_1 - a_{\text{lo}}b_{\text{hi}}$ must be a multiple of $2^{\lceil p/2 \rceil}$ because a_{lo} is an integer, and b_{hi} is a multiple of $2^{\lceil p/2 \rceil}$. Hence, Line 5 is computed exactly, and $err_2 = a_{\text{hi}}b_{\text{lo}} + a_{\text{lo}}b_{\text{lo}} + \text{err}(a \otimes b)$.

To show that Line 6 is computed without roundoff error, note that $a_{\text{lo}}b_{\text{lo}}$ is an integer no greater than 2^{p-1} (because a_{lo} and b_{lo} are integers no greater than $2^{\lceil p/2 \rceil - 1}$), and $\text{err}(a \otimes b)$ is an integer no greater than 2^{p-1}. Thus, $err_3 = a_{\text{lo}}b_{\text{lo}} + \text{err}(a \otimes b)$ is an integer no greater than 2^p, and is expressible in p bits.

Finally, Line 7 is exact simply because $y = -\text{err}(a \otimes b)$ can be expressed in p bits. Hence, $ab = x + y$.

If round-to-even tiebreaking is used, x and y are nonadjacent by analogy to Corollary 9. ■

2.6 Expansion Scaling

The following algorithm, which multiplies an expansion by a floating-point value, is the second key new result of this article.

Theorem 19 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, and let b be a p-bit value where $p \geq 4$. Suppose that the components of e are sorted in order of increasing magnitude, except that any of the e_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{2m} h_i = be$, where the components of h are also in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e is nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

```
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
```

As illustrated in Figure 13, SCALE-EXPANSION multiplies each component of e by b and sums the results. It should be apparent why the final expansion h is the desired product, but it is not so obvious why the components of h are guaranteed to be nonoverlapping and in increasing order. Two lemmata will aid the proof.

Lemma 20 *Let e_i and e_j be two nonoverlapping nonzero components of e, with $i < j$ and $|e_i| < |e_j|$. Let T_i be a correctly rounded approximation to $e_i b$, and let $T_i + t_i$ be a two-component expansion exactly equal to $e_i b$. (Such an expansion is produced by Line 3, but here is defined also for $i = 1$.) Then t_i is too small in magnitude to overlap the double-width product $e_j b$. Furthermore, if e_i and e_j are nonadjacent, then t_i is not adjacent to $e_j b$.*

Proof: By scaling e and b by appropriate powers of 2 (thereby shifting their exponents without changing their significands), one may assume without loss of generality that e_j and b are integers with magnitude less than 2^p, and that $|e_i| < 1$ (and hence a radix point falls between e_j and e_i).

It follows that $e_j b$ is an integer, and $|e_i b| < 2^p$. The latter fact and exact rounding imply that $|t_i| \leq \frac{1}{2}$. Hence, $e_j b$ and t_i do not overlap.

If e_i and e_j are nonadjacent, scale e so that e_j is an integer and $|e_i| < \frac{1}{2}$. Then $|t_i| \leq \frac{1}{4}$, so $e_j b$ and t_i are not adjacent. ■

Lemma 21 *For some i, let r be the smallest integer such that $|e_i| < 2^r$ (hence e_i does not overlap 2^r). Then $|Q_{2i}| \leq 2^r|b|$, and thus $|h_{2i-1}| \leq 2^{r-1}\text{ulp}(b)$.*

Proof: The inequality $|Q_{2i}| \leq 2^r|b|$ holds for $i = 1$ after Line 1 is executed even if Q_2 is rounded to a larger magnitude, because $|e_1 b| < 2^r|b|$, and $2^r|b|$ is expressible in p bits. For larger values of i, the bound is proven by induction. Assume that R is the smallest integer such that $|e_{i-1}| < 2^R$; by the inductive hypothesis, $|Q_{2i-2}| \leq 2^R|b|$.

Because e_i and e_{i-1} are nonoverlapping, e_i must be a multiple of 2^R. Suppose that r is the smallest integer such that $|e_i| < 2^r$; then $|e_i| \leq 2^r - 2^R$.

Lines 3, 4, and 5 compute Q_{2i}, an approximation of $Q_{2i-2} + e_i b$, and are subject to roundoff error in Lines 4 and 5. Suppose that Q_{2i-2} and $e_i b$ have the same sign, that $|Q_{2i-2}|$ has its largest possible value $2^R|b|$, and that $|e_i|$ has its largest possible value $2^r - 2^R$. For these assignments, roundoff does not occur in Lines 4 and 5, and $|Q_{2i}| = |Q_{2i-2} + e_i b| = 2^r|b|$. Otherwise, roundoff may occur, but the monotonicity of floating-point multiplication and addition ensures that $|Q_{2i}|$ cannot be larger than $2^r|b|$.

The inequality $|h_{2i-1}| \leq 2^{r-1}\text{ulp}(b)$ is guaranteed by exact rounding because h_{2i-1} is the roundoff term associated with the computation of Q_{2i} in Line 5. ■

Proof of Theorem 19: One can prove inductively that at the end of each iteration of the **for** loop, the invariant $Q_{2i} + \sum_{j=1}^{2i-1} h_j = \sum_{j=1}^{i} e_j b$ holds. Certainly this invariant holds for $i = 1$ after Line 1 is executed. By induction on Lines 3, 4, and 5, one can deduce that the invariant holds for all (relevant values of) i. (The use of FAST-TWO-SUM in Line 5 will be justified shortly.) Thus, after Line 6 is executed, $\sum_{j=1}^{2m} h_j = b \sum_{j=1}^{m} e_j$.

I shall prove that the components of h are nonoverlapping by showing that each time a component of h is written, that component is smaller than and does not overlap either the accumulator Q nor any of the remaining products ($e_j b$); hence, the component cannot overlap any portion of their sum. The first claim, that each component h_j does not overlap the accumulator Q_{j+1}, is true because h_j is the roundoff error incurred while computing Q_{j+1}.

To show that each component of h is smaller than and does not overlap the remaining products, I shall consider h_1, the remaining odd components of h, and the even components of h separately. The component h_1, computed by Line 1, does not overlap the remaining products $(e_2b, e_3b, \ldots)$ by virtue of Lemma 20. The even components, which are computed by Line 4, do not overlap the remaining products because, by application of Lemma 1 to Line 4, a component $|h_{2i-2}|$ is no larger than $|t_i|$, which is bounded in turn by Lemma 20.

Odd components of h, computed by Line 5, do not overlap the remaining products by virtue of Lemma 21, which guarantees that $|h_{2i-1}| \leq 2^{r-1}\text{ulp}(b)$. The remaining products are all multiples of $2^r\text{ulp}(b)$ (because the remaining components of e are multiples of 2^r).

If round-to-even tiebreaking is used, the output of each TWO-SUM, FAST-TWO-SUM, and TWO-PRODUCT statement is nonadjacent. If e is nonadjacent as well, the arguments above are easily modified to show that h is nonadjacent.

The use of FAST-TWO-SUM in Line 5 is justified because $|T_i| \geq |Q_{2i-1}|$ (except if $T_i = 0$, in which case FAST-TWO-SUM still works correctly). To see this, recall that e_i is a multiple of 2^R (with R defined as in Lemma 21), and consider two cases: if $|e_i| = 2^R$, then T_i is computed exactly and $t_i = 0$, so $|T_i| = 2^R|b| \geq |Q_{2i-2}| = |Q_{2i-1}|$. If $|e_i|$ is larger than 2^R, it is at least twice as large, and hence T_i is at least $2|Q_{2i-2}|$, so even if roundoff occurs and t_i is not zero, $|T_i| > |Q_{2i-2}| + |t_i| \geq |Q_{2i-1}|$.

Note that if an input component e_i is zero, then two zero output components are produced, and the accumulator value is unchanged ($Q_{2i} = Q_{2i-2}$). ■

The following corollary demonstrates that SCALE-EXPANSION is compatible with FAST-EXPANSION-SUM.

Corollary 22 *If e is strongly nonoverlapping and round-to-even tiebreaking is used, then h is strongly nonoverlapping.*

Proof: Because e is nonoverlapping, h is nonoverlapping by Theorem 19. We have also seen that if e is nonadjacent, then h is nonadjacent and hence strongly nonoverlapping; but e is only guaranteed to be strongly nonoverlapping, and may deviate from nonadjacency.

Suppose two successive components e_i and e_{i+1} are adjacent. By the definition of strongly nonoverlapping, e_i and e_{i+1} are both powers of two and are not adjacent to e_{i-1} or e_{i+2}. Let s be the integer satisfying $e_i = 2^s$ and $e_{i+1} = 2^{s+1}$. For these components the multiplication of Line 3 is exact, so $T_i = 2^s b$, $T_{i+1} = 2^{s+1}b$, and $t_i = t_{i+1} = 0$. Applying Lemma 1 to Line 4, $h_{2i-2} = h_{2i} = 0$. However, the components h_{2i-1} and h_{2i+1} may cause difficulty (see Figure 14). We know h is nonoverlapping, but can these two components be adjacent to their neighbors or each other?

The arguments used in Theorem 19 to prove that h is nonadjacent, if e is nonadjacent and round-to-even tiebreaking is used, can be applied here as well to show that h_{2i-1} and h_{2i+1} are not adjacent to any components of h produced before or after them, but they may be adjacent to each other. Assume that h_{2i-1} and h_{2i+1} are adjacent (they cannot be overlapping).

h_{2i+1} is computed in Line 5 from T_{i+1} and Q_{2i+1}. The latter addend is equal to Q_{2i}, because $t_{i+1} = 0$. Q_{2i} is not adjacent to h_{2i-1}, because they are produced in Line 5 from a FAST-TWO-SUM operation. Hence, the least significant nonzero bit of h_{2i+1} (that is, the bit that causes it to be adjacent to h_{2i-1}) must have come

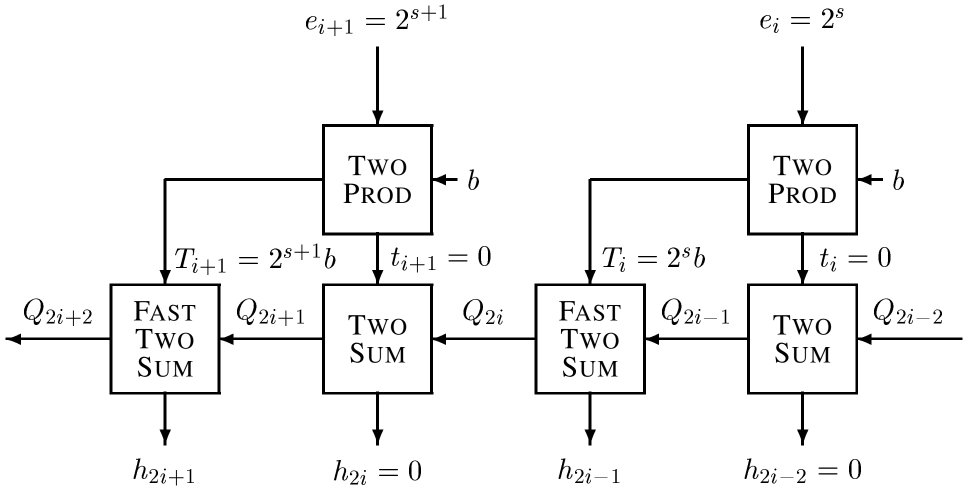

Figure 14: An adjacent pair of one-bit components in a strongly nonoverlapping input expansion may cause SCALE-EXPANSION to produce an adjacent pair of one-bit components in the output expansion.

from T_{i+1}, which is equal to $2^{s+1}b$. It follows that h_{2i+1} is a multiple of $2^{s+1}\text{ulp}(b)$. Because $|e_{i+1}| < 2^{s+2}$, Lemma 21 implies that $|h_{2i+1}| \leq 2^{s+1}\text{ulp}(b)$. Hence, $|h_{2i+1}| = 2^{s+1}\text{ulp}(b)$.

Similarly, because $|e_i| < 2^{s+1}$, Lemma 21 implies that $|h_{2i-1}| \leq 2^s\text{ulp}(b)$. The components h_{2i+1} and h_{2i-1} can only be adjacent in the case $|h_{2i-1}| = 2^s\text{ulp}(b)$. In this case, both components are expressible in one bit.

Hence, each adjacent pair of one-bit components in the input can give rise to an isolated adjacent pair of one-bit components in the output, but no other adjacent components may appear. If e is strongly nonoverlapping, so is h. ■

2.7 Compression and Approximation

The algorithms for manipulating expansions do not usually express their results in the most compact form. In addition to the interspersed zero components that have already been mentioned (and are easily eliminated), it is also common to find components that represent only a few bits of an expansion's value. Such fragmentation rarely becomes severe, but it can cause the largest component of an expansion to be a poor approximation of the value of the whole expansion; the largest component 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.

The COMPRESS algorithm below finds a compact form for an expansion. More importantly, COMPRESS guarantees that the largest component is a good approximation to the whole expansion. If round-to-even tiebreaking is used, COMPRESS also converts nonoverlapping expansions into nonadjacent expansions.

Priest [23] presents a more complicated "Renormalization" procedure that compresses optimally. Its greater running time is rarely justified by the marginal reduction in expansion length, unless there is a need to put expansions in a canonical form.

Theorem 23 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, where $m \geq 3$. Suppose that the components of e are sorted in order of increasing magnitude, except that any of the e_i may*

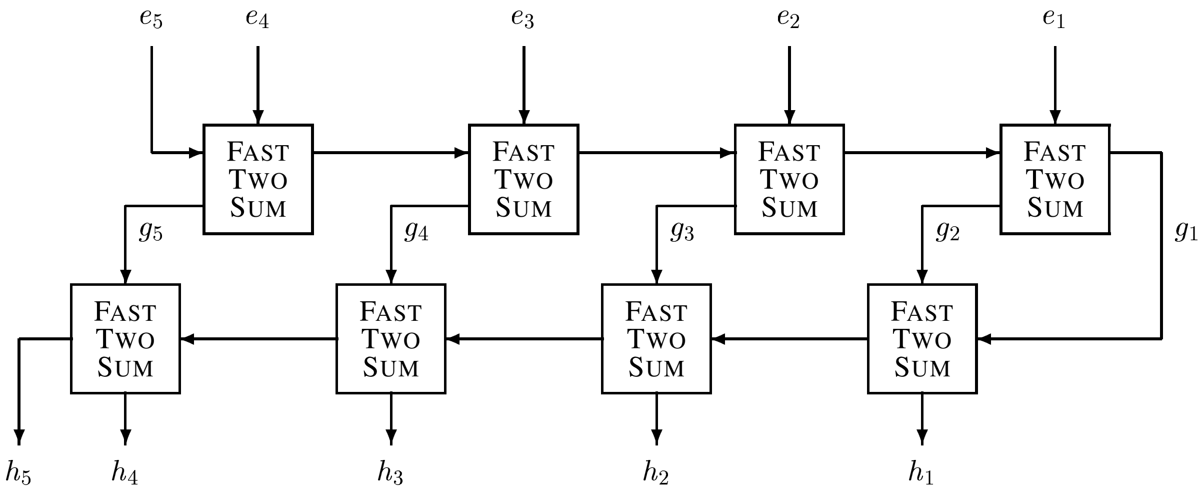

Figure 15: Operation of COMPRESS when no zero-elimination occurs.

be zero. Then the following algorithm will produce a nonoverlapping expansion h *(nonadjacent if round-to-even tiebreaking is used) such that* $h = \sum_{i=1}^{n} h_i = e$*, where the components* h_i *are in order of increasing magnitude. If* $h \neq 0$*, none of the* h_i *will be zero. Furthermore, the largest component* h_n *approximates* h *with an error smaller than* $\mathrm{ulp}(h_n)$.

```
COMPRESS(e)
    Q ⇐ e_m
    bottom ⇐ m
    for i ⇐ m − 1 downto 1
        (Q, q) ⇐ FAST-TWO-SUM(Q, e_i)
        if q ≠ 0 then
            g_bottom ⇐ Q
            bottom ⇐ bottom − 1
            Q ⇐ q
    g_bottom ⇐ Q
    top ⇐ 1
    for i ⇐ bottom + 1 to m
        (Q, q) ⇐ FAST-TWO-SUM(g_i, Q)
        if q ≠ 0 then
            h_top ⇐ Q
            top ⇐ top + 1
    h_top ⇐ Q
    Set n (the length of h) to top
    return h
```

Figure 15 illustrates the operation of COMPRESS. For clarity, g and h are presented as two separate arrays in the COMPRESS pseudocode, but they can be combined into a single working array without conflict by replacing every occurrence of "g" with "h".

Proof Sketch: COMPRESS works by traversing the expansion from largest to smallest component, then back from smallest to largest, replacing each adjacent pair with its two-component sum. The first traversal, from largest to smallest, does most of the compression. The expansion $g_m + g_{m-1} + \cdots + g_{bottom}$ produced by Lines 1 through 9 has the property that $g_{j-1} \leq \text{ulp}(g_j)$ for all j (and thus successive components overlap by at most one bit). This fact follows because the output of FAST-TWO-SUM in Line 4 has the property that $q \leq \frac{1}{2}\text{ulp}(Q)$, and the value of q thus produced can only be increased slightly by the subsequent addition of smaller nonoverlapping components.

The second traversal, from smallest to largest, clips any overlapping bits. The use of FAST-TWO-SUM in Line 12 is justified because the property that $g_{i-1} \leq \text{ulp}(g_i)$ guarantees that Q (the sum of the components that are smaller than g_i) is smaller than g_i. The expansion $h_{top} + h_{top-1} + \cdots + h_2 + h_1$ is nonoverlapping (nonadjacent if round-to-even tiebreaking is used) because FAST-TWO-SUM produces nonoverlapping (nonadjacent) output.

During the second traversal, an approximate total is maintained in the accumulator Q. The component h_{n-1} is produced by the last FAST-TWO-SUM operation that produces a roundoff term; this roundoff term is no greater than $\frac{1}{2}\text{ulp}(h_n)$. Hence, the sum $|h_{n-1} + h_{n-2} + \cdots + h_2 + h_1|$ (where the components of h are nonoverlapping) is less than $\text{ulp}(h_n)$, therefore $|h - h_n| < \text{ulp}(h_n)$. ■

To ensure that h_n is a good approximation to h, only the second traversal is necessary; however, the first traversal is more effective in reducing the number of components. The fastest way to approximate e is to simply sum its components from smallest to largest; by the reasoning used above, the result errs by less than one ulp. This observation is the basis for an APPROXIMATE procedure that is used in the predicates of Section 4.

Theorem 23 is not the strongest statement that can be made about COMPRESS. COMPRESS is effective even if the components of the input expansion have a certain limited amount of overlap. Furthermore, the bound for $|h - h_n|$ is not tight. (I conjecture that the largest possible relative error is exhibited by a number that contains a nonzero bit every pth bit; observe that $1 + \frac{1}{2}\text{ulp}(1) + \frac{1}{4}[\text{ulp}(1)]^2 + \cdots$ cannot be further compressed.) These improvements complicate the proof and are not explored here.

2.8 Other Operations

Distillation is the process of summing k unordered p-bit values. Distillation can be performed by the divide-and-conquer algorithm of Priest [23], which uses any expansion addition algorithm to sum the values in a tree-like fashion as illustrated in Figure 16. Each p-bit addend is a leaf of the tree, and each interior node represents a call to an expansion addition algorithm. If EXPANSION-SUM is used (and zero elimination is not), then it does not matter whether the tree is balanced; distillation will take precisely $\frac{1}{2}k(k-1)$ TWO-SUM operations, regardless of the order in which expansions are combined. If FAST-EXPANSION-SUM is used, the speed of distillation depends strongly on the balance of the tree. A well-balanced tree will yield an $\mathcal{O}(k \log k)$ distillation algorithm, an asymptotic improvement over distilling with EXPANSION-SUM. As I have mentioned, it is usually fastest to use an unrolled EXPANSION-SUM to create expansions of length four, and FAST-EXPANSION-SUM with zero elimination to sum these expansions.

To find the product of two expansions e and f, use SCALE-EXPANSION (with zero elimination) to form the expansions $ef_1, ef_2, \ldots$, then sum these using a distillation tree.

Division cannot always, of course, be performed exactly, but it can be performed to arbitrary precision by an iterative algorithm that employs multiprecision addition and multiplication. Consult Priest [23] for one such algorithm.

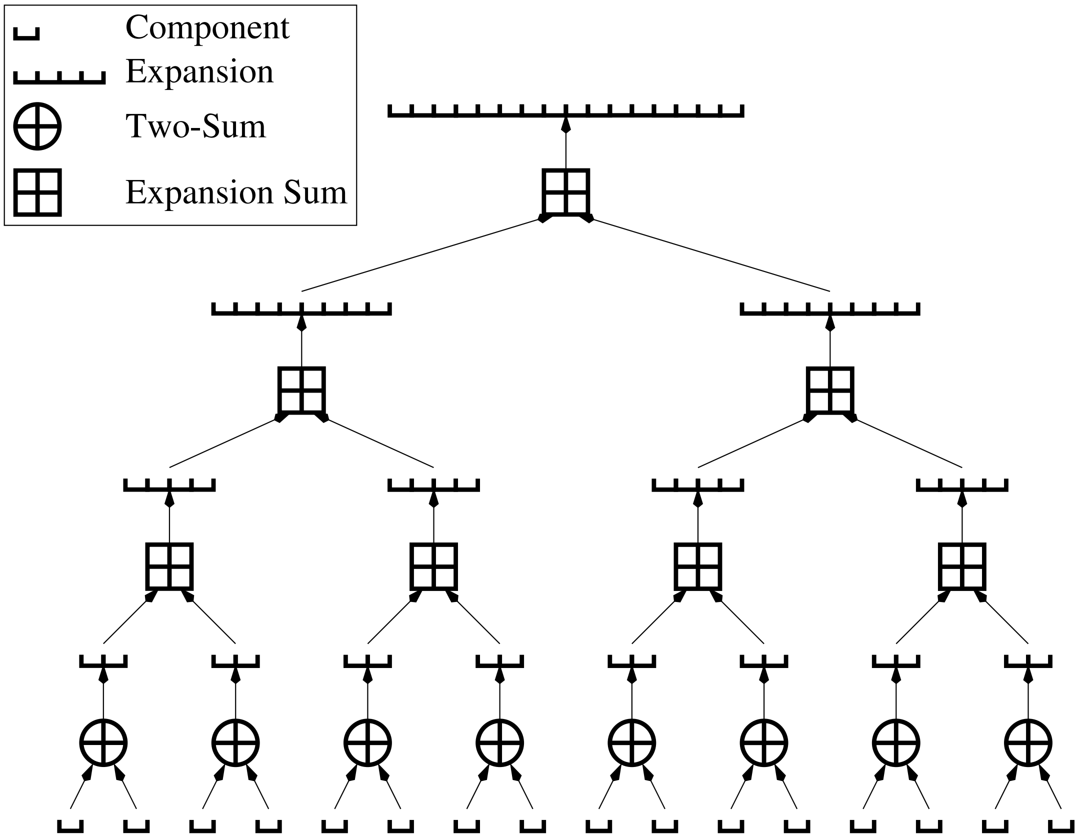

Figure 16: Distillation of sixteen p-bit floating-point values.

The easiest way to compare two expansions is to subtract one from the other, and test the sign of the result. An expansion's sign can be easily tested because of the nonoverlapping property; simply check the sign of the expansion's most significant nonzero component. (If zero elimination is employed, check the component with the largest index.) A nonoverlapping expansion is equal to zero if and only if all its components are equal to zero.

3 Adaptive Precision Arithmetic

3.1 Why Adaptivity?

Exact arithmetic is expensive, and when it can be avoided, it should be. Some applications do not need exact results, but require the absolute error of a result to fall below some threshold. If this threshold is known before the computation is performed, it is economical to employ *adaptivity by prediction*. One writes several procedures, each of which approximates the result with a different degree of precision, and with a correspondingly different speed. Error bounds are derived for each of these procedures; these bounds are typically much cheaper to compute than the approximations themselves, except for the least precise approximation. For any particular input, the application computes the error bounds and uses them to choose the procedure that will attain the necessary accuracy most cheaply.

Sometimes, however, one cannot determine whether a computation will be accurate enough before it is done. An example is when one wishes to bound the relative error, rather than the absolute error, of the result. (A special case is determining the sign of an expression; the result must have relative error less than one.) The result may prove to be much larger than its error bound, and low precision arithmetic will suffice, or it may be so close to zero that it is necessary to evaluate it exactly to satisfy the bound on relative error. One cannot generally know in advance how much precision is needed.

In the context of determinant evaluation for computational geometry, Fortune and Van Wyk [11] suggest using a floating-point filter. An expression is evaluated approximately in hardware precision arithmetic first. Forward error analysis determines whether the approximate result can be trusted; if not, an exact result is computed. If the exact computation is only needed occasionally, the application is slowed only a little.

One might hope to improve this idea further by computing a sequence of 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 often possible to use intermediate results as stepping stones to more accurate results; work already done is not discarded but is refined.

3.2 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 17(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, as illustrated in Figure 17(b). Expand the expression to form a polynomial.

In the expanded expression, the terms containing many occurrences of y variables (roundoff errors) are dominated by terms containing fewer occurrences. As an example, consider the expression $(a_x - b_x)^2 + (a_y - b_y)^2$ (Figure 17), 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

$$(x_1^2 + x_2^2) + (2x_1y_1 + 2x_2y_2) + (y_1^2 + y_2^2). \quad (5)$$

It is significant that each y_i is small relative to its corresponding x_i. Using standard terminology from forward error analysis [28], the quantity $\frac{1}{2}\text{ulp}(1)$ is called the *machine epsilon*, denoted ϵ. Recall that exact rounding guarantees that $|y_i| \leq \epsilon|x_i|$; the quantity ϵ bounds the *relative error* $\text{err}(a \circledast b)/(a \circledast b)$ of any basic floating-point operation. Note that $\epsilon = 2^{-p}$. In IEEE 754 double precision arithmetic, $\epsilon = 2^{-53}$; in single precision, $\epsilon = 2^{-24}$.

Expression 5 can be divided into three parts, having magnitudes of $\mathcal{O}(1)$, $\mathcal{O}(\epsilon)$, and $\mathcal{O}(\epsilon^2)$, respectively. Denote these parts T_0, T_1, and T_2. More generally, for any expression expanded in this manner, let T_i be the sum of all products containing i of the y variables, so that T_i has magnitude $\mathcal{O}(\epsilon^i)$.

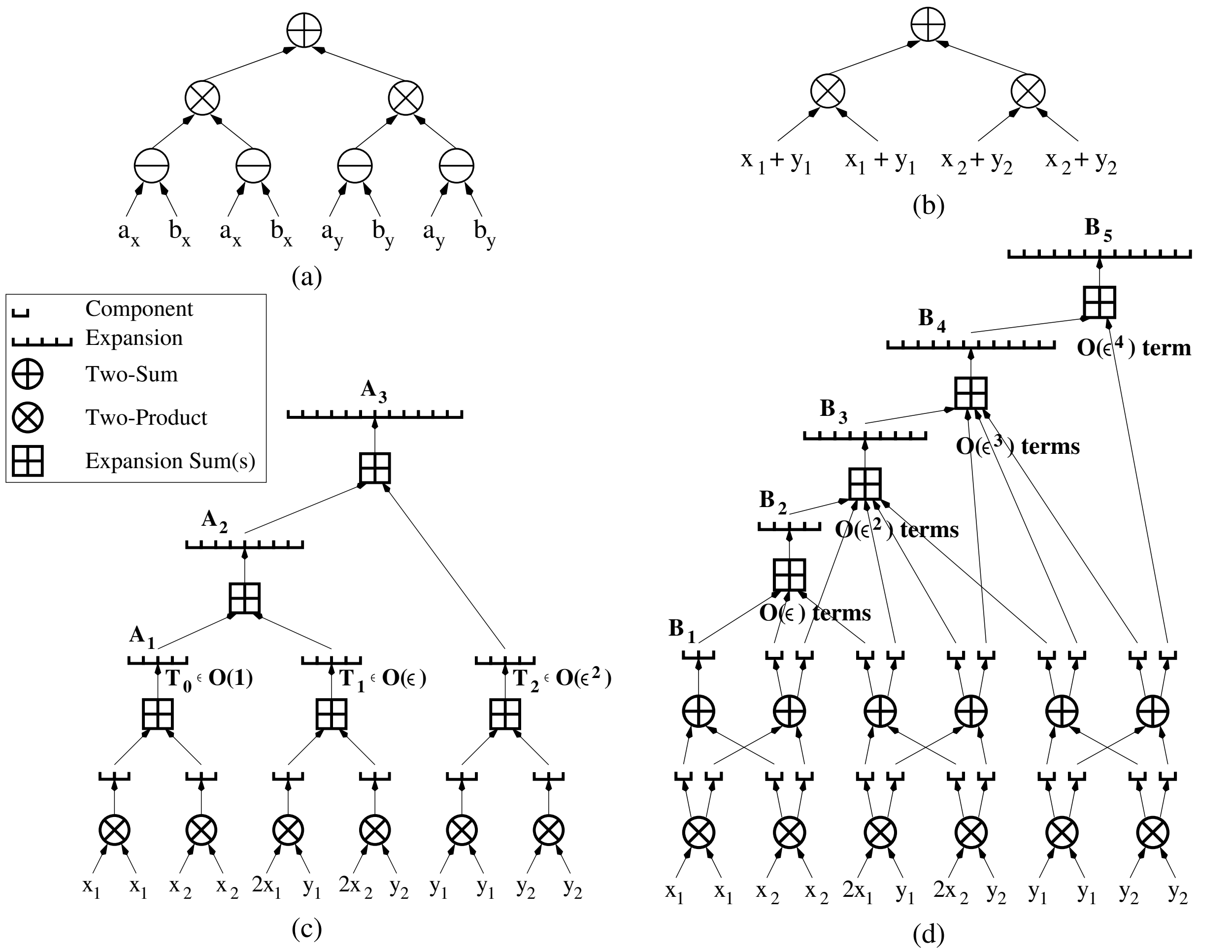

Figure 17: (a) Formula for the square of the distance between two points a and b. (b) The lowest subexpressions in the tree are expressed as the sum of an approximate value and a roundoff error. (c) A simple incremental adaptive method for evaluating the expression. The approximations A_1 and A_2 are generated and tested in turn. The final expansion A_3 is exact. Each A_i includes all terms of size $\mathcal{O}(\epsilon^{i-1})$ or larger, and hence has error no greater than $\mathcal{O}(\epsilon^i)$. (d) Incremental adaptivity taken to an extreme. The three subexpression trees T_0, T_1, and T_2 are themselves calculated adaptively. Each B_i incorporates only the terms needed to reduce its error to $\mathcal{O}(\epsilon^i)$.

One can obtain an approximation A_j with error no larger than $\mathcal{O}(\epsilon^j)$ by computing exactly the sum of the first j terms, T_0 through T_{j-1}. The sequence $A_1, A_2, \ldots$ of increasingly accurate approximations can be formed incrementally; A_j is the exact sum of A_{j-1} and T_{j-1}. Members of this sequence are generated and tested, as illustrated in Figure 17(c), until one is sufficiently accurate.

The approximation A_j is not the way to achieve an error bound of $\mathcal{O}(\epsilon^j)$ with the least amount of work. For instance, a floating-point calculation of $(x_1^2 + x_2^2)$ using no exact arithmetic techniques will achieve an $\mathcal{O}(\epsilon)$ error bound, albeit with a larger constant than the error bound for A_1. Experimentation has shown that the fastest adaptive predicates are written by calculating an approximation having bound $\mathcal{O}(\epsilon^j)$ as quickly as possible, then moving on to the next smaller order of magnitude. Improvements in the constant prefacing each error bound will make a difference in only a small number of cases. Hence, I will consider two modifications to the technique just described. The first modification computes each error bound from the minimum possible number of roundoff terms. This lazy approach is presented here for instructional purposes, but is not generally the fastest. The second modification I will consider, and the one I recommend for use, is faster because it spends less time collating small data.

The first modification is to compute the subexpressions T_0, T_1, and T_2 adaptively as well. The method is the same: replace each bottom-level subexpression of T_0 (and T_1 and T_2) with the sum of an approximate result and an error term, and expand T_0 into a sum of terms of differing order. An approximation B_j having an error bound of magnitude $\mathcal{O}(\epsilon^j)$ may be found by approximating each T term with error $\mathcal{O}(\epsilon^j)$. Because the term T_k has magnitude at most $\mathcal{O}(\epsilon^k)$, it need not be approximated with any better relative error than $\mathcal{O}(\epsilon^{j-k})$.

Figure 17(d) shows that the method is as lazy as possible, in the sense that each approximation B_j uses only the roundoff terms needed to obtain an $\mathcal{O}(\epsilon^j)$ error bound. (Note that this is true at every level of the tree. It is apparent in the figure that every roundoff term produced is fed into a different calculation than the larger term produced with it.) However, the laziest approach is not necessarily the fastest approach. The cost of this method is unnecessarily large for two reasons. First, recall from Section 2.8 that FAST-EXPANSION-SUM is most effective when terms are summed in a balanced manner. The additions in Figure 17(d) are less well balanced than those in Figure 17(c). Second, and more importantly, there is a good deal of overhead for keeping track of many small pieces of the sum; the method sacrifices most of the advantages of the compressed form in which expansions are represented. Figure 17(d) does not fully reveal how convoluted this extreme form of adaptivity can become for larger expressions. In addition to having an unexpectedly large overhead, this method can be exasperating for the programmer.

The final method for incremental adaptivity I shall present, which is used to derive the geometric predicates in Section 4, falls somewhere between the two described above. As in the first method, compute the sequence $A_1, A_2, \ldots$, and define also $A_0 = 0$. We have seen that the error bound of each term A_j may be improved from $\mathcal{O}(\epsilon^j)$ to $\mathcal{O}(\epsilon^{j+1})$ by (exactly) adding T_j to it. However, because the magnitude of T_j itself is $\mathcal{O}(\epsilon^j)$, the same effect can be achieved (with a slightly worse constant in the error bound) by computing T_j with floating-point arithmetic and tolerating the roundoff error, rather than computing T_j exactly. Hence, an approximation C_{j+1} having an $\mathcal{O}(\epsilon^{j+1})$ error bound is computed by summing A_j and an inexpensive *correctional term*, which is merely the floating-point approximation to T_j, as illustrated in Figure 18. C_{j+1} is nearly as accurate as A_{j+1} but takes much less work to compute. If C_{j+1} is not sufficiently accurate, then it is thrown away, and the exact value of T_j is computed and added to A_j to form A_{j+1}. This scheme reuses the work done in performing exact calculations, but does not reuse the correctional terms. (In practice, no speed can be gained by reusing the correctional terms.)

The first value (C_1) computed by this method is an approximation to T_0; if C_1 is sufficiently accurate, it is unnecessary to compute the y terms, or use any exact arithmetic techniques, at all. (Recall that the y terms are more expensive to compute than the x terms.) 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. It is slower when the exact result must be computed, but is faster in applications that rarely need an exact result. In some cases, it may be desirable to test certain members of both sequences A and C for accuracy; the predicates defined in Section 4 do so.

All three methods of making expressions adaptive are mechanical and can be automated. An expression compiler that converts expressions into code that evaluates these expressions adaptively, with automatically computed error bounds, would be valuable. (Fortune and Van Wyk [12] have produced such a compiler for integer operands, using straightforward floating-point filters instead of the more complicated adaptive methods described here. Their expression compiler is discussed in the next section.)

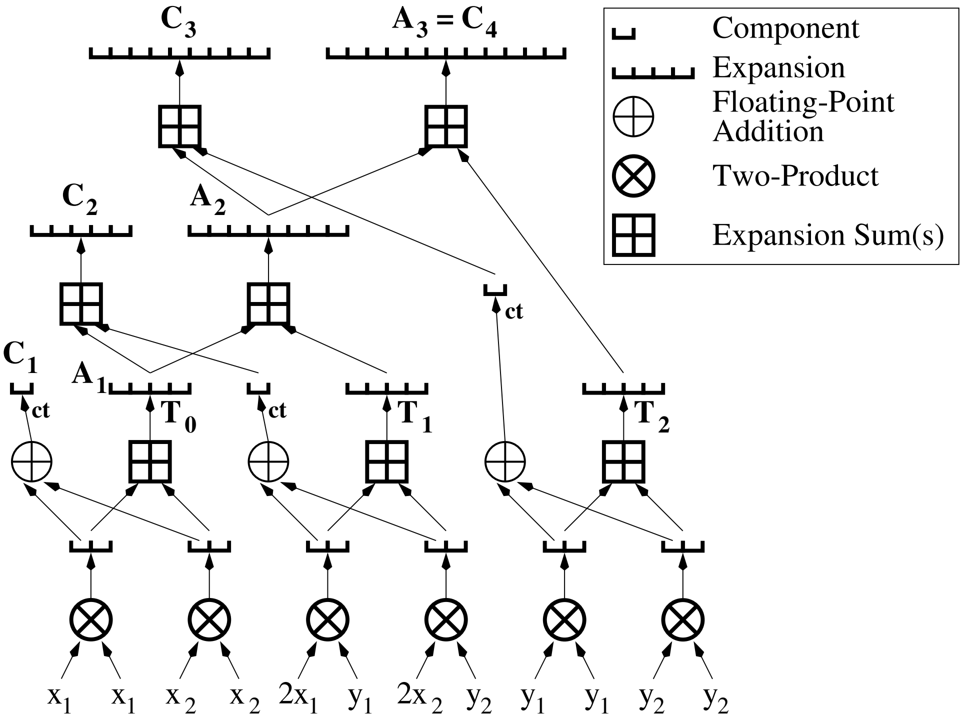

Figure 18: An adaptive method of intermediate complexity that is frequently more efficient than the other two. Each C_i achieves an $\mathcal{O}(\epsilon^i)$ error bound by adding an inexpensive correctional term (labeled "ct") to A_{i-1}.

The reader may wonder if writing an expression in sum-of-products form isn't inefficient. In ordinary floating-point arithmetic it often is, but it seems to make little difference when using the exact arithmetic algorithms of Section 2. Indeed, the multiplication operation described in Section 2.8 multiplies two expansions by expanding the product into sum-of-products form.

These ideas are not exclusively applicable to the multiple-component approach to arbitrary precision arithmetic. They will work with multiple-digit formats as well, though the details differ.

4 Implementation of Geometric Predicates

4.1 Related Work in Robust Computational Geometry

Most geometric algorithms are not originally designed for robustness at all; they are based on the *real RAM model*, in which quantities are allowed to be arbitrary real numbers, and all arithmetic is exact. There are several ways a geometric algorithm that is correct within the real RAM model can go wrong in an encounter with roundoff error. The output might be incorrect, but be correct for some perturbation of its input. The result might be usable yet not be valid for any imaginable input. Or, the program may simply crash or fail to produce a result. To reflect these possibilities, geometric algorithms are divided into several classes with varying amounts of robustness: *exact algorithms*, which are always correct; *robust algorithms*, which are always correct for some perturbation of the input; *stable algorithms*, for which the perturbation is small; *quasi-robust algorithms*, whose results might be geometrically inconsistent, but nevertheless satisfy some weakened consistency criterion; and *fragile algorithms*, which are not guaranteed to produce any usable output at all. The next several pages are devoted to a discussion of representative research in each class, and of the circumstances in which exact arithmetic and other techniques are or are not applicable. For more extensive surveys of geometric robustness, see Fortune [9] and Hoffmann [15].

Exact algorithms. A geometric algorithm is *exact* if it is guaranteed to produce a correct result when given an exact input. (Of course, the input to a geometric algorithm may only be an approximation of some real-world configuration, but this difficulty is ignored here.) Exact algorithms use exact arithmetic in some form, whether in the form of a multiprecision library or in a more disguised form.

There are several exact arithmetic schemes designed specifically for computational geometry; most are methods for exactly evaluating the sign of a determinant, and hence can be used to perform the orientation and incircle tests. Clarkson [6] 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[6].

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 [12, 11] propose a more general approach (not specific to determinants, or even to predicates) that represents integers using a standard multiple-digit 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 is run; an expression can be used to generate several functions, each for arguments of different bit lengths. Because the expression and the bit lengths of all operands are fixed in advance, LN can tune the exact arithmetic aggressively, eliminating loops, function calls, and memory management. The running time of a function produced by LN depends on the bit complexity of the inputs. Fortune and Van Wyk report an order-of-magnitude speed improvement over the use of multiprecision libraries (for equal bit complexity). Furthermore, LN gains another speed improvement by installing floating-point filters wherever appropriate, calculating error bounds automatically.

Karasick, Lieber, and Nackman [16] 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 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.

[6]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."

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 with 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 2). I am not aware of any prior literature on exact determinant evaluation that considers floating-point operands, except for one limited example: Ottmann, Thiemt, and Ullrich [22] advocate the use of an *accurate scalar product* operation, ideally implemented in hardware (though the software-level distillation algorithm described in Section 2.8 may also be used), as a way to evaluate some predicates such as the 2D orientation test.

Exact determinant algorithms do not satisfy the needs of all applications. A program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored. If the intersections may themselves become endpoints of lines that generate more intersections, then intersections of greater and greater bit complexity may be generated. Even exact rational arithmetic is not sufficient for all applications; a solid modeler, for instance, might need to determine the vertices of the intersection of two independent solids that have been rotated through arbitrary angles. Yet exact floating-point arithmetic can't even cope with rotating a square 45° in the plane, because irrational vertex coordinates result. The problem of constructed irrational values has been partly attacked by the implementation of "real" numbers in the LEDA library of algorithms [4]. Values derived from square roots (and other arithmetic operations) are stored in symbolic form when necessary. Comparisons with such numbers are resolved with great numerical care, albeit sometimes at great cost; separation bounds are computed where necessary to ensure that the sign of an expression is determined accurately. Floating-point filters and another form of adaptivity (approximating a result repeatedly, doubling the precision each time) are used as well.

For the remainder of this discussion, consideration is restricted to algorithms whose input is geometric (e.g. coordinates are specified) but whose output is purely combinatorial, such as the construction of a convex hull or an arrangement of hyperplanes.

Robust algorithms. There are algorithms that can be made correct with straightforward implementations of exact arithmetic, but suffer an unacceptable loss of speed. An alternative is to relax the requirement of a correct solution, and instead accept a solution that is "close enough" in some sense that depends upon the application. Without exact arithmetic, an algorithm must somehow find a way to produce sensible output despite the fact that geometric tests occasionally tell it lies. No general techniques have emerged yet, although bandages have appeared for specific algorithms, usually ensuring robustness or quasi-robustness through painstaking design and error analysis. The lack of generality of these techniques is not the only limitation of the relaxed approach to robustness; there is a more fundamental difficulty that deserves careful discussion.

When disaster strikes and a real RAM-correct algorithm implemented in floating-point arithmetic fails to produce a meaningful result, it is often because the algorithm has performed tests whose results are mutually contradictory. Figure 19 shows an error that arose in a two-dimensional Delaunay triangulation program I wrote. The program, which employs a divide-and-conquer algorithm presented by Guibas and Stolfi [14], failed in a subroutine that merges two triangulations into one. The geometrically nonsensical triangulation in the illustration was produced.

On close inspection with a debugger, I found that the failure was caused by a single incorrect result of the incircle test. At the bottom of Figure 19 appear four nearly collinear points whose deviation from collinearity has been greatly exaggerated for clarity. The points a, b, c, and d had been sorted by their

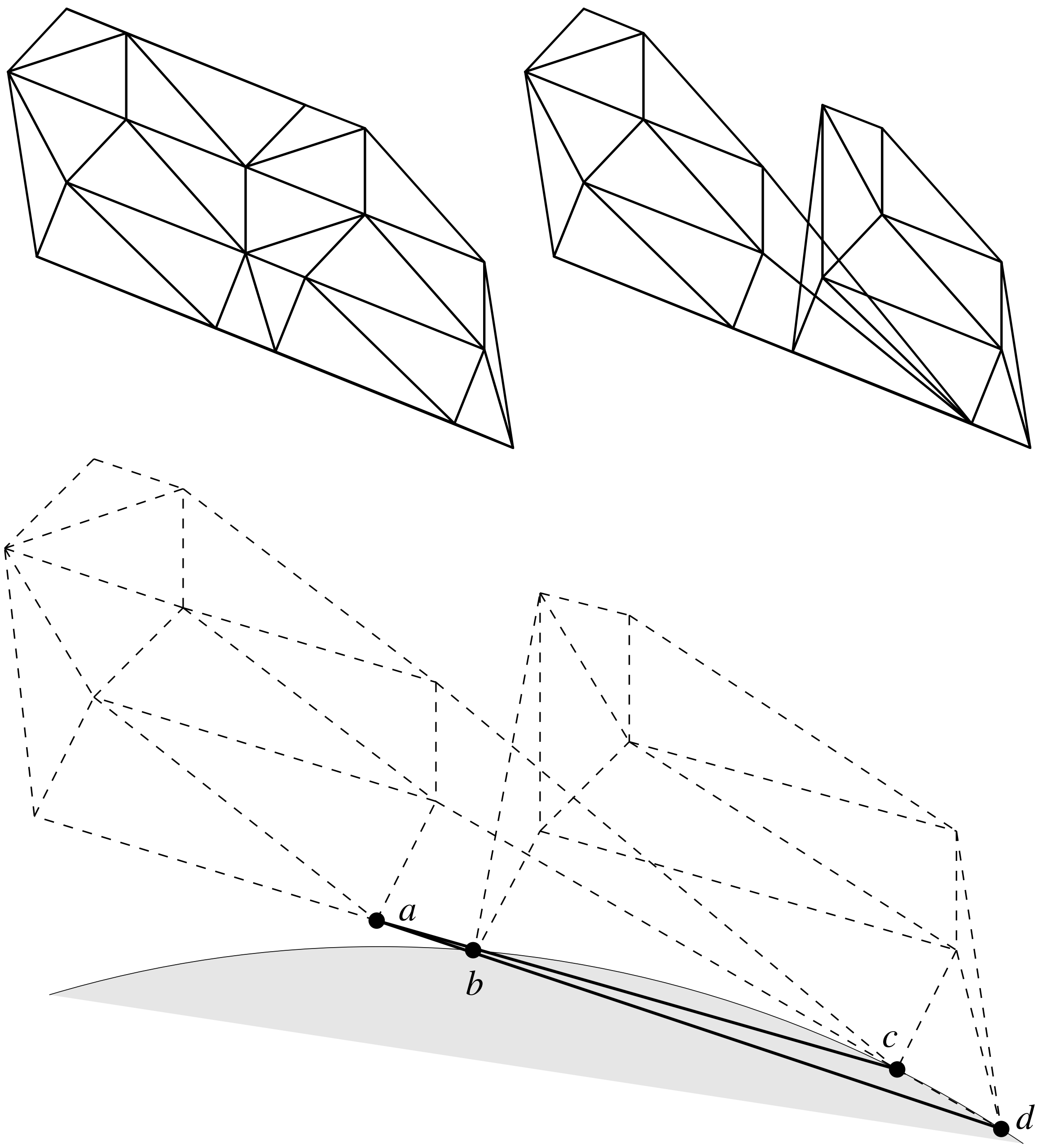

Figure 19: Top left: A Delaunay triangulation. Top right: An invalid triangulation created due to roundoff error. Bottom: Exaggerated view of the inconsistencies that led to the problem. The algorithm "knew" that the point b lay between the lines ac and ad, but an incorrect incircle test claimed that a lay inside the circle dcb.

x-coordinates, and b had been correctly established (by orientation tests) to lie below the line ac and above the line ad. In principle, a program could deduce from these facts that a cannot fall inside the circle dcb. Unfortunately, the incircle test incorrectly declared that a lay inside, thereby leading to the invalid result.

It is significant that the incircle test was not just wrong about these particular points; it was inconsistent with the "known combinatorial facts." A correct algorithm (that computes a purely combinatorial result) will produce a meaningful result if its test results are wrong but are consistent with each other, because there exists an input for which those test results are correct. Following Fortune [8], an algorithm is *robust* if it always produces the correct output under the real RAM model, and under approximate arithmetic always produces an output that is consistent with some hypothetical input that is a perturbation of the true input; it is *stable* if this perturbation is small. Typically, bounds on the perturbation are proven by backward error analysis. Using only approximate arithmetic, Fortune gives an algorithm that computes a planar convex hull that is correct for points that have been perturbed by a relative error of at most $\mathcal{O}(\epsilon)$ (where ϵ is defined as in Section 3.2), and an algorithm that maintains a triangulation that can be made planar by perturbing each vertex by a relative error of at most $\mathcal{O}(n^2\epsilon)$, where n is the number of vertices. If it seems surprising that a "stable" algorithm cannot keep a triangulation planar, consider the problem of inserting a new vertex so close to an existing edge that it is difficult to discern which side of the edge the vertex falls on. Only exact arithmetic can prevent the possibility of creating an "inverted" triangle.

One might wonder if my triangulation program can be made robust by avoiding any test whose result can be inferred from previous tests. Fortune [8] explains that

> [a]n algorithm is *parsimonious* if it never performs a test whose outcome has already been determined as the formal consequence of previous tests. A parsimonious algorithm is clearly robust, since any path through the algorithm must correspond to some geometric input; making an algorithm parsimonious is the most obvious way of making it robust. In principle it is possible to make an algorithm parsimonious: since all primitive tests are polynomial sign evaluations, the question of whether the current test is a logical consequence of previous tests can be phrased as a statement of the existential theory of the reals. This theory is at least NP-hard and is decidable in polynomial space [5]. Unfortunately, the full power of the theory seems to be necessary for some problems. An example is the *line arrangement problem*: given a set of lines (specified by real coordinates (a, b, c), so that $ax + by = c$), compute the combinatorial structure of the resulting arrangement in the plane. It follows from recent work of Mnev [21] that the problem of deciding whether a combinatorial arrangement is actually realizable with lines is as hard as the existential theory of the reals. Hence a parsimonious algorithm for the line arrangement problem . . . seems to require the solution of NP-hard problems.

Because exact arithmetic does not require the solution of NP-hard problems, an intermediate course is possible; one could employ parsimony whenever it is efficient to do so, and resort to exact arithmetic otherwise. Consistency is guaranteed if exact tests are used to bootstrap the "parsimony engine." I am not aware of any algorithms in the literature that take this approach, although geometric algorithms are often designed by their authors to avoid the more obviously redundant tests.

Quasi-robust algorithms. The difficulty of determining whether a line arrangement is realizable suggests that, without exact arithmetic, robustness as defined above may be an unattainable goal. However, sometimes one can settle for an algorithm whose output might not be realizable. I place such algorithms in a bag labeled with the fuzzy term *quasi-robust*, which I apply to any algorithm whose output is somehow provably distinguishable from nonsense. Milenkovic [20] circumvents the aforementioned NP-hardness result while using approximate arithmetic by constructing pseudo-line arrangements; a *pseudo-line* is a curve constrained to lie very close to an actual line. Fortune [10] presents a 2D Delaunay triangulation algorithm that constructs, using approximate arithmetic, a triangulation that is nearly Delaunay in a well-defined sense using the pseudo-line-like notion of pseudocircles. Unfortunately, the algorithm's running time is $\mathcal{O}(n^2)$, which compares poorly with the $\mathcal{O}(n \log n)$ time of optimal algorithms. Milenkovic's and Fortune's algorithms are both *quasi-stable*, having small error bounds. Milenkovic's algorithm can be thought of as a quasi-robust algorithm for line arrangements, or as a robust algorithm for pseudo-line arrangements.

Barber [3] pioneered an approach in which uncertainty, including the imprecision of input data, is a part of each geometric entity. *Boxes* are structures that specify the location and the uncertainty in location of a vertex, edge, facet, or other geometric structure. Boxes may arise either as input or as algorithmic constructions; any uncertainty resulting from roundoff error is incorporated into their shapes and sizes. Barber presents algorithms for solving the point-in-polygon problem and for constructing convex hulls in any dimension. For the point-in-polygon problem, "can't tell" is a valid answer if the uncertainty inherent in the input or introduced by roundoff error prevents a sure determination. The salient feature of Barber's Quickhull convex hull algorithm is that it merges hull facets that cannot be guaranteed (through error analysis) to be clearly locally convex. The *box complex* produced by the algorithm is guaranteed to contain the true convex hull, bounding it, if possible, both from within and without.

The degree of robustness required of an algorithm is typically determined by how its output is used. For instance, many point location algorithms can fail when given a non-planar triangulation. For this very reason, my triangulator crashed after producing the flawed triangulation in Figure 19.

The reader should take three lessons from this section. First, problems due to roundoff can be severe and difficult to solve. Second, even if the inputs are imprecise and the user isn't picky about the accuracy of the output, internal consistency may still be necessary if any output is to be produced at all; exact arithmetic may be required even when exact results aren't. Third, neither exact arithmetic nor clever handling of tests that tell falsehoods is a universal balm. However, exact arithmetic is attractive when it is applicable, because it can be employed by naïve program developers without the time-consuming need for careful analysis of a particular algorithm's behavior when faced with imprecision. (I occasionally hear of implementations where more than half the developers' time is spent solving problems of roundoff error and degeneracy.) Hence, efforts to improve the speed of exact arithmetic in computational geometry are well justified.

4.2 The Orientation and Incircle Tests

Let a, b, c, and d be four points in the plane. 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 the points are in clockwise order, and zero if the points 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 these 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. Both ORIENT2D and INCIRCLE have the symmetry property that interchanging any two of their parameters reverses the sign of their result.

These definitions extend trivially 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. (One can apply a *left-hand* *rule*: orient your left hand with fingers curled to follow the circular sequence abc. If your thumb points toward d, ORIENT3D returns a positive value.) To generalize the orientation test to dimensionality d, let $u_1, u_2, \ldots, u_d$ be the unit vectors; ORIENT is defined so that $\text{ORIENT}(u_1, u_2, \ldots, u_d, 0) = 1$.

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

$$\text{ORIENT3D}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{6}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \tag{7}$$

$$\text{INSPHERE}(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2 + e_y^2 + e_z^2 & 1 \end{vmatrix} \tag{8}$$

$$= \begin{vmatrix} a_x - e_x & a_y - e_y & a_z - e_z & (a_x - e_x)^2 + (a_y - e_y)^2 + (a_z - e_z)^2 \\ b_x - e_x & b_y - e_y & b_z - e_z & (b_x - e_x)^2 + (b_y - e_y)^2 + (b_z - e_z)^2 \\ c_x - e_x & c_y - e_y & c_z - e_z & (c_x - e_x)^2 + (c_y - e_y)^2 + (c_z - e_z)^2 \\ d_x - e_x & d_y - e_y & d_z - e_z & (d_x - e_x)^2 + (d_y - e_y)^2 + (d_z - e_z)^2 \end{vmatrix} \tag{9}$$

These formulae generalize to other dimensions in the obvious way. Expressions 6 and 7 can be shown to be equivalent by simple algebraic transformations, as can Expressions 8 and 9 with a little more effort. These equivalences are unsurprising because one expects the result of any orientation or incircle test not to change if all the points undergo an identical translation in the plane. Expression 7, for instance, follows from Expression 6 by translating each point by $-d$.

When computing these determinants using the techniques of Section 2, the choice between Expressions 6 and 7, or between 8 and 9, is not straightforward. In principle, Expression 6 seems preferable because it can only produce a 96-component expansion, whereas Expression 7 could produce an expansion having 192 components. These numbers are somewhat misleading, however, because with zero-elimination, expansions rarely grow longer than six components in real applications. Nevertheless, Expression 7 takes roughly 25% more time to compute in exact arithmetic, and Expression 9 takes about 50% more time than Expression 8. 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 7 and 9 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 6 and 8 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 for Expressions 7 and 9 are generally much smaller than for Expressions 6 and 8. Furthermore, the translation can often be done without roundoff error. Figure 20 demonstrates a toy

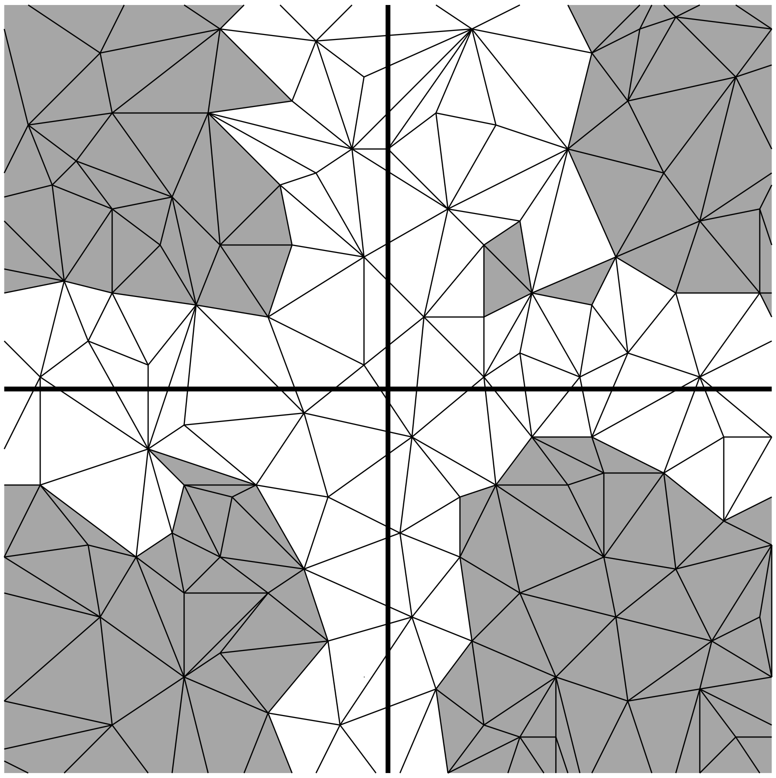

Figure 20: Shaded triangles can be translated to the origin without incurring roundoff error (Lemma 5). In most triangulations, such triangles are the common case.

problem: suppose ORIENT2D is used to find the orientation of each triangle in a triangulation. Thanks to Lemma 5, 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) will 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 number of methods are surveyed by Fortune and Van Wyk [11], and only their conclusion is repeated here. The cheapest method of evaluating the determinant of a 5×5 or smaller matrix seems to be by 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 three columns, and so on. All four of my predicates use this method.

4.3 ORIENT2D

My implementation of ORIENT2D computes a sequence of up to four results (labeled A through D) as illustrated in Figure 21. 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 floating-point approximation A, which is computed

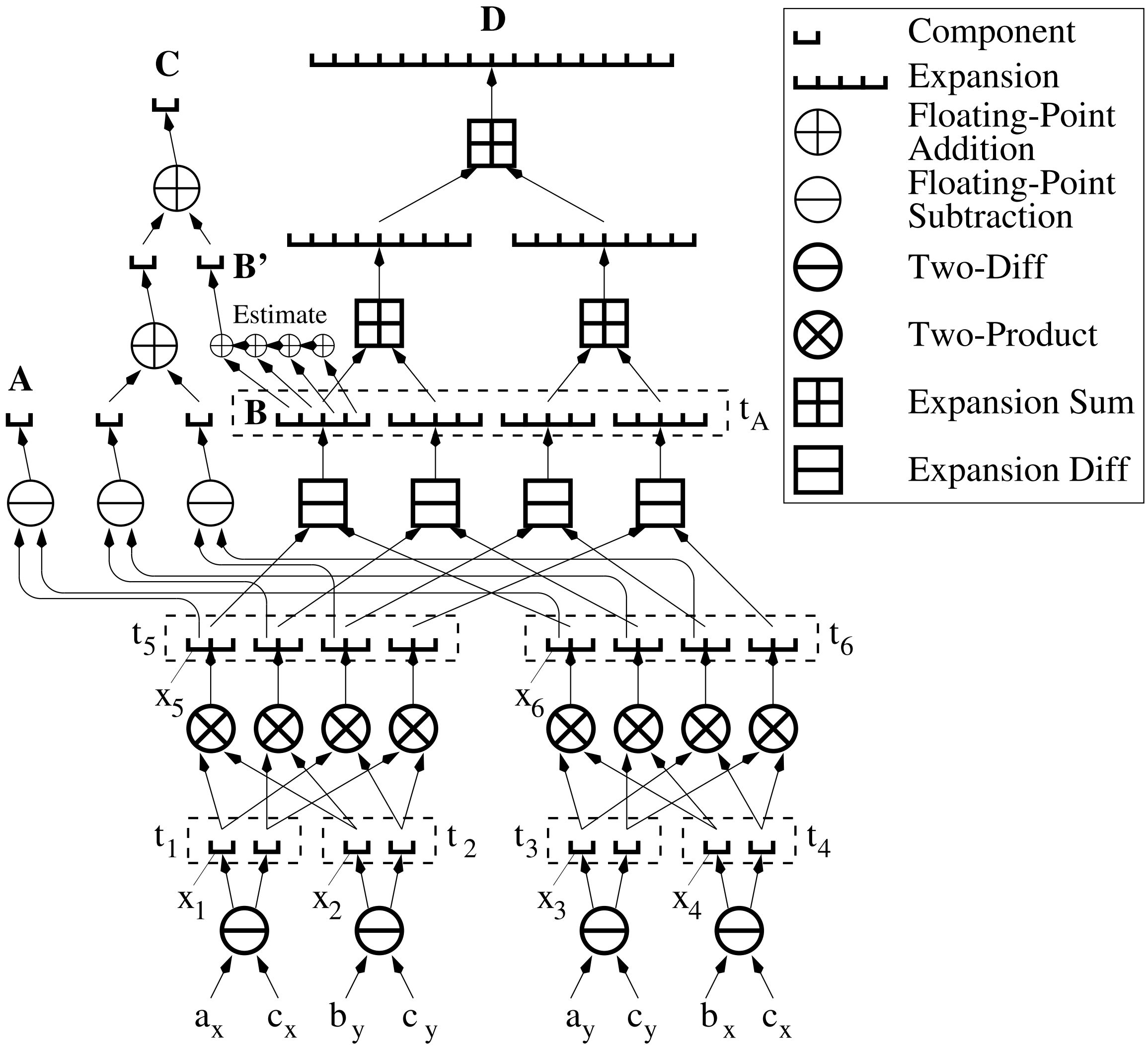

Figure 21: Adaptive calculations used by the 2D orientation test. Dashed boxes represent nodes in the original expression tree.

without resort to any exact arithmetic techniques. Although the four-component expansion B, like A, has an error of $\mathcal{O}(\epsilon)$, it is an appropriate value to test because B is the exact result if the four subtractions at the bottom of the expression tree are performed without roundoff error (corresponding to the shaded triangles in Figure 20). Because this is the common case, ORIENT2D explicitly tests for it; execution continues only if roundoff occurred during the translation of coordinates and B is smaller than its error bound. 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 unusual 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 2.7 is used to find 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 $\mathcal{O}(\epsilon B)$, which 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 a, b, and c; the determinant defined by these points is zero. If a coordinate of one of these points is perturbed by a single ulp, 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 the predicates considered herein, on both random and "practical" point sets. Determinants that don't stop with approximation C are nearly always zero.

The derivation of error bounds for these values is tricky, so an example is given here. The easiest way to apply forward error analysis to an expression whose value is calculated in floating-point arithmetic is to express the exact value of each subexpression in terms of the computed value plus an unknown error term whose magnitude is bounded. For instance, the error incurred by the computation $x \Leftarrow a \oplus b$ is no larger than $\epsilon|x|$. Furthermore, the error is smaller than $\epsilon|a+b|$. Each of these bounds is useful under different circumstances. If t represents the true value $a+b$, an abbreviated way of expressing these notions is to write $t = x \pm \epsilon|x|$ and $t = x \pm \epsilon|t|$. Henceforth, this notation will be used as shorthand for the relation $t = x + \lambda$ for some λ that satisfies $|\lambda| \leq \epsilon|x|$ and $|\lambda| \leq \epsilon|t|$.

Let us consider the error bound for A. For each subexpression in the expression tree of the orientation test, denote its true (exact) value t_i and its approximate value x_i as follows.

$$\begin{array}{ll} t_1 = a_x - c_x & x_1 = a_x \ominus c_x \\ t_2 = b_y - c_y & x_2 = b_y \ominus c_y \\ t_3 = a_y - c_y & x_3 = a_y \ominus c_y \\ t_4 = b_x - c_x & x_4 = b_x \ominus c_x \\ t_5 = t_1 t_2 & x_5 = x_1 \otimes x_2 \\ t_6 = t_3 t_4 & x_6 = x_3 \otimes x_4 \\ t_A = t_5 - t_6 & \mathrm{A} = x_5 \ominus x_6 \end{array}$$

From these definitions, it is clear that $t_1 = x_1 \pm \epsilon|x_1|$; similar bounds hold for t_2, t_3, and t_4. Observe also that $x_5 = x_1 \otimes x_2 = x_1 x_2 \pm \epsilon|x_5|$. It follows that

$$\begin{aligned} t_5 = t_1 t_2 &= x_1 x_2 \pm (2\epsilon + \epsilon^2)|x_1 x_2| \\ &= x_5 \pm \epsilon|x_5| \pm (2\epsilon + \epsilon^2)(|x_5| \pm \epsilon|x_5|) \\ &= x_5 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|x_5|. \end{aligned}$$

Similarly, $t_6 = x_6 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|x_6|$.

It may seem odd to be keeping track of terms smaller than $\mathcal{O}(\epsilon)$, but the effort to find the smallest machine-representable coefficient for each error bound is justified if it ever prevents a determinant computation from becoming more expensive than necessary. An error bound for A can now be derived.

$$\begin{aligned} t_A = t_5 - t_6 &= x_5 - x_6 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|) \\ &= \mathrm{A} \pm \epsilon|\mathrm{A}| \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|) \end{aligned}$$

One can minimize the effect of the term $\epsilon|\mathrm{A}|$ by taking advantage of the fact that we are only interested in the sign of t_A. One can conclude with certainty that A has the correct sign if

$$(1 - \epsilon)|\mathrm{A}| > (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|),$$

Approximation	Error bound
A	$(3\epsilon + 16\epsilon^2) \otimes (\lvert x_5 \rvert \oplus \lvert x_6 \rvert)$
B′	$(2\epsilon + 12\epsilon^2) \otimes (\lvert x_5 \rvert \oplus \lvert x_6 \rvert)$
C	$(3\epsilon + 8\epsilon^2) \otimes \lvert \mathrm{B}' \rvert \oplus (9\epsilon^2 + 64\epsilon^3) \otimes (\lvert x_5 \rvert \oplus \lvert x_6 \rvert)$

Table 1: Error bounds for the expansions calculated by ORIENT2D. B′ is a p-bit approximation of the expansion B, computed by the APPROXIMATE procedure. Note that each coefficient is expressible in p bits.

which is true if

$$|\mathrm{A}| \geq (3\epsilon + 6\epsilon^2 + 8\epsilon^3)(|x_5| + |x_6|).$$

This bound is not directly applicable, because its computation will incur roundoff error. To account for this, multiply the coefficient by $(1+\epsilon)^2$ (a factor of $(1+\epsilon)$ for the addition of $|x_5|$ and $|x_6|$, and another such factor for the multiplication). Hence, we are secure that the sign of A is correct if

$$|\mathrm{A}| \geq (3\epsilon + 12\epsilon^2 + 24\epsilon^3) \otimes (|x_5| \oplus |x_6|).$$

This bound is not directly applicable either, because the coefficient is not expressible in p bits. Rounding up to the next p-bit number, we have the coefficient $(3\epsilon + 16\epsilon^2)$, which should be exactly computed once at program initialization and reused during each call to ORIENT2D.

Error bounds for A, B′, and C are given in Table 1. The bound for B′ takes advantage of Theorem 23, which shows that B′ approximates B with relative error less than 2ϵ. (Recall from Section 2.7 that the largest component of B might have only one bit of precision.)

These bounds have the pleasing property that they are 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 (A) in the two cases that occur most commonly in practice.

Compiler effects affect the implementation of ORIENT2D. By separating the calculation of A and the remaining calculations into two procedures, with the former calling the latter if necessary, I reduced the time to compute A by 25%, presumably because of improvements in the compiler's ability to perform register allocation.

Table 2 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 4.5 will demonstrate. It is significant that for 53-bit integer inputs, the multiple-stage predicates will rarely pass stage B because the initial translation is usually

Double precision ORIENT2D timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Collinear
Approximate (7)	0.15	0.15	0.16
Exact (6)	6.56	6.89	6.31
Exact (7)	8.35	8.48	8.13
Exact (6), MPFUN	92.85	94.03	84.97
Adaptive A (7), approximate	0.28	0.27	0.22
Adaptive B (7)			1.89
Adaptive C (7)			2.14
Adaptive D (7), exact			8.35
LN adaptive (7), approximate	0.32	n/a	
LN adaptive (7), exact		n/a	4.43

Table 2: Timings for ORIENT2D on a DEC 3000/700 with a 225 MHz Alpha processor. All determinants use the 2D version of either Expression 6 or the more stable Expression 7 as indicated. The first two columns indicate input points generated from a uniform random distribution and a geometric random distribution. The third column considers two points chosen from one of the random distributions, and a third point chosen to be approximately collinear to the first two. Timings for the adaptive tests are categorized according to which result was the last generated. Each timing is an average of 60 or more randomly generated inputs. For each such input, time was measured by a Unix system call before and after 10,000 iterations of the predicate. Individual timings vary by approximately 10%. Timings of Bailey's MPFUN package and Fortune and Van Wyk's LN package are included for comparison.

done without roundoff error; hence, the LN-generated ORIENT2D usually takes more than twice as long to produce an exact result. It should be emphasized, however, that these are not inherent differences between LN's multiple-digit integer approach and my multiple-component floating-point approach; LN could, in principle, employ the same runtime error estimate and a similar multiple-stage adaptivity scheme.

4.4 ORIENT3D, INCIRCLE, and INSPHERE

Figure 22 illustrates the implementation of ORIENT3D, which is similar to the ORIENT2D implementation. 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 $\mathcal{O}(\epsilon^2)$. D is the exact determinant.

Error bounds for the largest component of each of these expansions are given in Table 3, partly in terms of the variables x_1, x_6, and x_7 in Figure 22. The 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 instances of coplanarity.

Table 4 lists timings for ORIENT3D, given random inputs. The error bound for A is expensive to compute, and increases the amount of time required to perform the approximate test in the adaptive case by a factor of two and a half. The gap between my exact algorithm and MPFUN is smaller than in the 2D case, but is still a factor of nearly eight.

Oddly, the table reveals that D is calculated more quickly than the exact result is calculated by the non-adaptive version of ORIENT3D. The explanation is probably that D is only computed when the determinant is zero or very close to zero, hence the lengths of the intermediate expansions are smaller than usual, and the

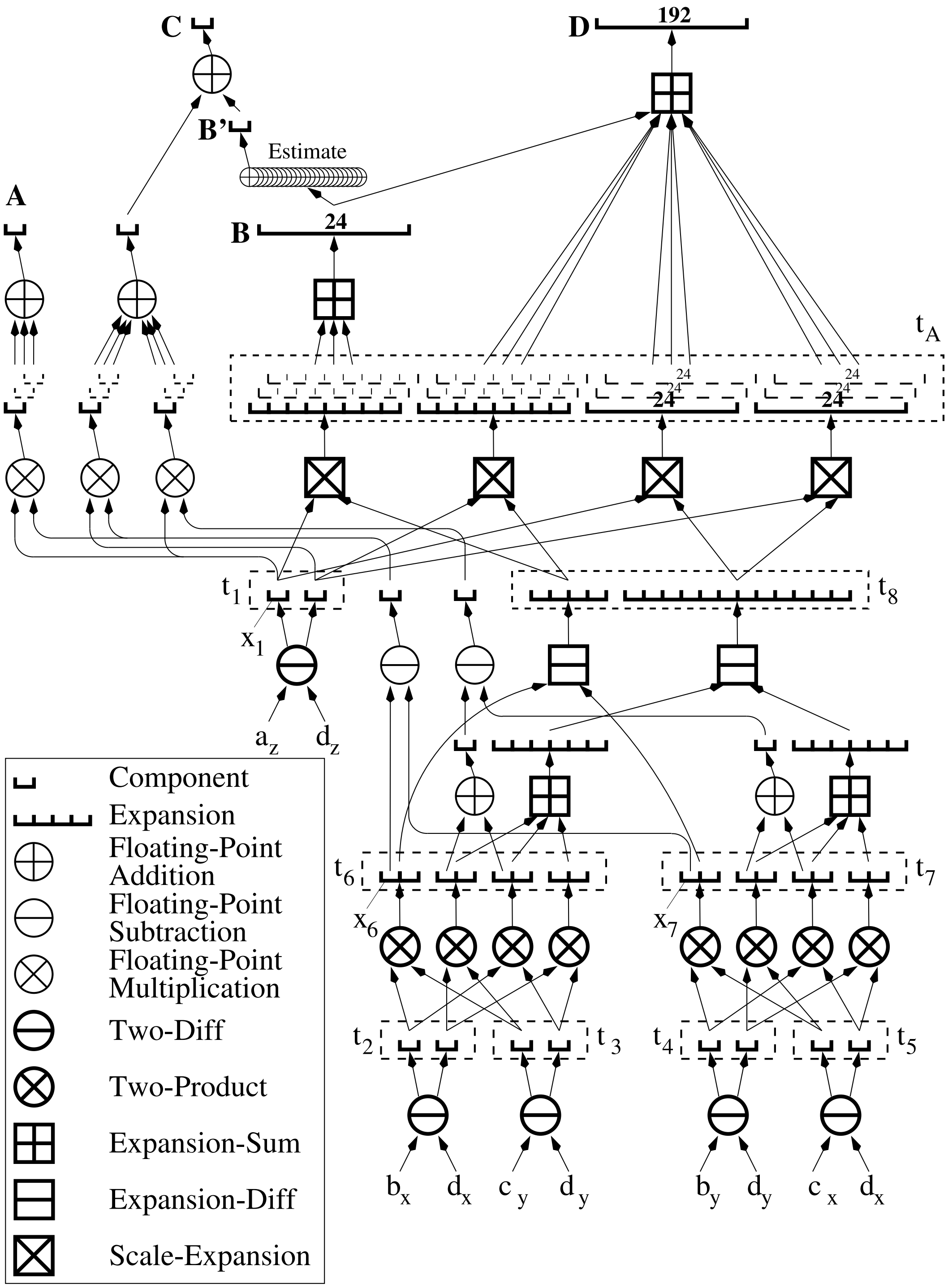

Figure 22: Adaptive calculations used by the 3D orientation test. Bold numbers indicate the length of an expansion. Only part of the expression tree is shown; two of the three cofactors are omitted, but their results appear as dashed components and expansions.

Approximation	Error bound
A	$(7\epsilon + 56\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
B′	$(3\epsilon + 28\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
C	$(3\epsilon + 8\epsilon^2) \otimes \lvert B'\rvert \oplus (26\epsilon^2 + 288\epsilon^3) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$

$$\begin{aligned}
\alpha_a &= |x_1| \otimes (|x_6| \oplus |x_7|) \\
&= |a_z \ominus d_z| \otimes (|(b_x \ominus d_x) \otimes (c_y \ominus d_y)| \oplus |(b_y \ominus d_y) \otimes (c_x \ominus d_x)|) \\
\alpha_b &= |b_z \ominus d_z| \otimes (|(c_x \ominus d_x) \otimes (a_y \ominus d_y)| \oplus |(c_y \ominus d_y) \otimes (a_x \ominus d_x)|) \\
\alpha_c &= |c_z \ominus d_z| \otimes (|(a_x \ominus d_x) \otimes (b_y \ominus d_y)| \oplus |(a_y \ominus d_y) \otimes (b_x \ominus d_x)|)
\end{aligned}$$

Table 3: Error bounds for the expansions calculated by ORIENT3D.

Double precision ORIENT3D timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Coplanar
Approximate (7)	0.25	0.25	0.25
Exact (6)	33.30	38.54	32.90
Exact (7)	42.69	48.21	42.41
Exact (6), MPFUN	260.51	262.08	246.64
Adaptive A (7), approximate	0.61	0.60	0.62
Adaptive B (7)			12.98
Adaptive C (7)			15.59
Adaptive D (7), exact			27.29
LN adaptive (7), approximate	0.85	n/a	
LN adaptive (7), exact		n/a	18.11

Table 4: Timings for ORIENT3D on a DEC 3000/700. All determinants are Expression 6 or the more stable Expression 7 as indicated. Each timing is an average of 120 or more randomly generated inputs. For each such input, time was measured by a Unix system call before and after 10,000 iterations of the predicate. Individual timings vary by approximately 10%.

computation time is less. Furthermore, when some of the point coordinates are translated without roundoff error, the adaptive predicate ignores branches of the expression tree that evaluate to zero.

INCIRCLE is implemented similarly to ORIENT3D, as the determinants are similar. The corresponding error bounds appear in Table 5, and timings appear in Table 6.

Timings for INSPHERE appear in Table 7. This implementation differs from the other 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. Fortunately, C is usually accurate enough.

The LN exact tests have an advantage of a factor of roughly 2.5 for INCIRCLE and 4 for INSPHERE, so the cost of handling floating-point operands is greater with the larger expressions. As with the orientation tests, this cost is mediated by better error bounds and four-stage adaptivity.

The timings for the exact versions of all four predicates show some sensitivity to the distribution of the

Approximation	Error bound
A	$(10\epsilon + 96\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
B′	$(4\epsilon + 48\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
C	$(3\epsilon + 8\epsilon^2) \otimes \lvert\mathrm{B}'\rvert \oplus (44\epsilon^2 + 576\epsilon^3) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$

$$\begin{aligned}
\alpha_a &= ((a_x \ominus d_x)^2 \oplus (a_y \ominus d_y)^2) \otimes (|(b_x \ominus d_x) \otimes (c_y \ominus d_y)| \oplus |(b_y \ominus d_y) \otimes (c_x \ominus d_x)|) \\
\alpha_b &= ((b_x \ominus d_x)^2 \oplus (b_y \ominus d_y)^2) \otimes (|(c_x \ominus d_x) \otimes (a_y \ominus d_y)| \oplus |(c_y \ominus d_y) \otimes (a_x \ominus d_x)|) \\
\alpha_c &= ((c_x \ominus d_x)^2 \oplus (c_y \ominus d_y)^2) \otimes (|(a_x \ominus d_x) \otimes (b_y \ominus d_y)| \oplus |(a_y \ominus d_y) \otimes (b_x \ominus d_x)|)
\end{aligned}$$

Table 5: Error bounds for the expansions calculated by INCIRCLE. Squares are approximate.

Double precision INCIRCLE timings in microseconds			
Method / Points	Uniform Random	Geometric Random	Nearly Cocircular
Approximate (9)	0.31	0.28	0.30
Exact (8)	71.66	83.01	75.34
Exact (9)	91.71	118.30	104.44
Exact (8), MPFUN	350.77	343.61	348.55
Adaptive A (9), approximate	0.64	0.59	0.64
Adaptive B (9)			44.56
Adaptive C (9)			48.80
Adaptive D (9), exact			78.06
LN adaptive (9), approximate	1.33	n/a	
LN adaptive (9), exact		n/a	32.44

Table 6: Timings for INCIRCLE on a DEC 3000/700. All determinants are the 2D version of either Expression 8 or the more stable Expression 9 as indicated. Each timing is an average of 100 or more randomly generated inputs, except adaptive stage D. (It is difficult to generate cases that reach stage D.) For each such input, time was measured by a Unix system call before and after 1,000 iterations of the predicate. Individual timings vary by approximately 10%.

operands; they take 5% to 30% longer to execute with geometrically distributed operands (whose exponents vary widely) than with uniformly distributed operands. This difference occurs because the intermediate and final expansions are larger when the operands have broadly distributed exponents. The exact orientation predicates are cheapest when their inputs are collinear/coplanar, because of the smaller expansions that result, but this effect does not occur for the exact incircle predicates.

4.5 Performance in Two Triangulation Programs

To evaluate the effectiveness of the adaptive tests in applications, I tested them in two of my Delaunay triangulation codes. Triangle [25] is a 2D Delaunay triangulator and mesh generator, publicly available from Netlib, that uses a divide-and-conquer algorithm [18, 14]. Pyramid is a 3D Delaunay tetrahedralizer that uses an incremental algorithm [27]. 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

Double precision INSPHERE timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Cospherical
Approximate (9)	0.93	0.95	0.93
Exact (8)	324.22	378.94	347.16
Exact (9)	374.59	480.28	414.13
Exact (8), MPFUN	1,017.56	1,019.89	1,059.87
Adaptive A (9), approximate	2.13	2.14	2.14
Adaptive B (9)			166.21
Adaptive C (9)			171.74
Adaptive D (8), exact			463.96
LN adaptive (9), approximate	2.35	n/a	
LN adaptive (9), exact		n/a	116.74

Table 7: Timings for INSPHERE on a DEC 3000/700. All determinants are Expression 8 or the more stable Expression 9 as indicated. Each timing is an average of 25 or more randomly generated inputs, except adaptive stage D. For each such input, time was measured by a Unix system call before and after 1,000 iterations of the predicate. Individual timings vary by approximately 10%.

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

The results for 2D, which appear in Table 8, 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 precisely for the point sets that are most likely to cause difficulties when exact arithmetic is not available.

The results for 3D, outlined in Table 9, are 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.

In these programs (and likely in any program), three of the four-stage predicates (INSPHERE being the exception) are faster than their LN equivalents. This is a surprise, considering that the four-stage predicates accept 53-bit floating-point inputs whereas the LN-generated predicates are restricted to 53-bit integer inputs. However, the integer predicates would probably outperform the floating-point predicates if they were to adopt the same runtime error estimate and a similar four-stage adaptivity scheme.

5 Caveats

Unfortunately, the arbitrary precision arithmetic routines described herein are not universally portable; both hardware and compilers can prevent them from functioning correctly.

2D divide-and-conquer Delaunay triangulation			
	Uniform Random	Perimeter of Circle	Tilted Grid
Input sites	1,000,000	1,000,000	1,000,000
ORIENT2D calls			
Adaptive A, approximate	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, approximate	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 8: Statistics for 2D divide-and-conquer Delaunay triangulation of several point sets. Timings are accurate to within 10%.

Compilers can interfere by making invalid optimizations based on misconceptions about floating-point arithmetic. For instance, a clever but incorrect compiler might cause expansion arithmetic algorithms to fail by deriving the "fact" that b_{virtual}, computed by Line 2 of FAST-TWO-SUM, is equal to b, and optimizing the subtraction away. This optimization would be valid if computers stored arbitrary real numbers, but is incorrect for floating-point numbers. Unfortunately, not all compiler developers are aware of the importance of maintaining correct floating-point language semantics, but as a whole, they seem to be improving. Goldberg [13, §3.2.3] presents several related examples of how carefully designed numerical algorithms can be utterly ruined by incorrect optimizations.

Even floating-point units that use binary arithmetic with exact rounding, including those that conform to the IEEE 754 standard, can have subtle properties that undermine the assumptions of the algorithms. The most common such difficulty is the presence of extended precision internal floating-point registers, such as those on the Intel 80486 and Pentium processors. While such registers usually improve the stability of floating-point calculations, they cause the methods described herein for determining the roundoff of an operation to fail. There are several possible workarounds for this problem. In C, it is possible to designate a variable as volatile, implying that it must be stored to memory. This ensures that the variable is rounded

3D incremental Delaunay tetrahedralization			
	Uniform Random	Surface of Sphere	Tilted Grid
Input sites	10,000	10,000	10,000
ORIENT3D calls			
Adaptive A, approximate	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 calls			
Adaptive A, approximate	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 9: Statistics for 3D incremental Delaunay tetrahedralization of several point sets. Timings are accurate to within 10%. The approximate code failed to terminate on the tilted grid input.

to a p-bit significand before it is used in another operation. Forcing intermediate values to be stored to memory and reloaded can slow down the algorithms significantly, and there is a worse consequence. Even a volatile variable could be *doubly rounded*, being rounded once to the internal extended precision format, then rounded again to single or double precision when it is stored to memory. The result after double rounding is not always the same as it would be if it had been correctly rounded to the final precision, and Priest [24, page 103] describes a case wherein the roundoff error produced by double rounding may not be expressible in p bits. This might be alleviated by a more complex (and slower) version of FAST-TWO-SUM. A better solution is to configure one's processor to round internally to double precision. While most processors with internal extended precision registers can be thus configured, and most compilers provide support for manipulating processor control state, such support varies between compilers and is not portable. Nevertheless, the speed advantage of multiple-component methods makes it well worth the trouble to learn the right incantation to correctly configure your processor.

The algorithms do work correctly without special treatment on most current Unix workstations. Nevertheless, users should be careful when trying the routines, or moving to a new platform, to ensure that the underlying assumptions of the method are not violated.

6 Conclusions

The algorithms presented herein are simple and fast; looking at Figure 8, it is difficult to imagine how expansions could be summed with fewer operations without special hardware assistance. Two features of these techniques account for the improvement in speed relative to other techniques, especially for numbers whose precision is only a few components in length. The first is the relaxation of the usual condition that numbers be normalized to fixed digit positions. Instead, one enforces the much weaker condition that expansions be nonoverlapping (or strongly nonoverlapping). Expansions can be summed and the resulting components made nonoverlapping at a cost of six floating-point operations and one comparison per component. It seems unlikely that normalization to fixed digit positions can be done so quickly in a portable way on current processors. The second feature to which I attribute the improved speed is the fact that most packages require expensive conversions between ordinary floating-point numbers and the packages' internal formats. With the techniques Priest and I describe, no conversions are necessary.

The reader may be misled and attribute the whole difference between my algorithms and MPFUN to the fact that I store double precision components, while MPFUN stores single precision digits, and imagine the difference would go away if MPFUN were reimplemented in double precision. Such a belief betrays a misunderstanding of how MPFUN works. MPFUN uses double precision arithmetic internally, and obtains exact results by using digits narrow enough that they can be multiplied exactly. Hence, MPFUN's half-precision digits are an integral part of its approach: to calculate exactly by avoiding roundoff error. The surprise of multiple-component methods is that reasonable speed can be attained by allowing roundoff to happen, then accounting for it after the fact.

As well as being fast, multiple-component algorithms are also reasonably portable, making no assumptions other than that a machine has binary arithmetic with exact rounding (and round-to-even tiebreaking if FAST-EXPANSION-SUM is to be used instead of LINEAR-EXPANSION-SUM). No representation-dependent tricks like bit-masking to extract exponent fields are used. There are still machines that cannot execute these algorithms correctly, but their numbers seem to be dwindling as the IEEE standard becomes entrenched.

Perhaps the greatest limitation of the multiple-component approach is that while it easily extends the precision of floating-point numbers, there is no simple way to extend the exponent range without losing much of the speed. The obvious approach, associating a separate exponent field with each component, is sure to be too slow. A more promising approach is to express each multiprecision number as a *multiexpansion* consisting of digits of very large radix, where each digit is an expansion coupled with an exponent. In this scheme, the true exponent of a component is the sum of the component's own exponent and the exponent of the expansion that contains it. The fast algorithms described in this article can be used to add or multiply individual digits; digits are normalized by standard methods (such as those used by MPFUN). IEEE double precision values have an exponent range of -1022 to 1023, so one could multiply digits of radix 2^{1000} with a simple expansion multiplication algorithm, or digits of radix 2^{2000} with a slightly more complicated one that splits each digit in half before multiplying.

The C code I have made publicly available might form the beginning of an extensive library of arithmetic routines similar to MPFUN, but a great deal of work remains to be done. In addition to the problem of expanding the exponent range, there is one problem that is particular to the multiple-component approach: it is not possible to use FFT-based multiplication algorithms without first renormalizing each expansion to a multiple-digit form. This normalization is not difficult to do, but it costs time and puts the multiple-component method at a disadvantage relative to methods that keep numbers in digit form as a matter of course.

As Priest points out, multiple-component 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 lies not with arbitrary precision libraries, but 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. If an algorithm can be made significantly more stable by using double or quadruple precision for a few key values, it may save a researcher from spending a great deal of time devising and analyzing a stabler algorithm; Priest [24, §5.1] offers several examples. 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 LN [12, 11], which converts an expression into exact arithmetic code, complete with error bound derivation and floating-point filters. Such a tool could also automate the process of breaking an expression into adaptive stages as described in Section 3.

To see how adaptivity can be used for more than just determining the sign of an expression, suppose one wishes to find, with relative error no greater than 1%, the center d of a circle that passes through the three points a, b, and c. One may use the following expressions.

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

The denominator of these fractions is precisely the expression computed by ORIENT2D. The computation of d is unstable if a, b, and c are nearly collinear; roundoff error in the denominator can dramatically change the result, or cause a division by zero. Disaster can be avoided, and the desired error bound enforced, by computing the denominator with a variant of ORIENT2D that accepts an approximation only if its relative error is roughly half of one percent. A similar adaptive routine could accurately compute the numerators.

It might be fruitful to explore whether the methods described by Clarkson [6] 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 matrices larger than 5×5), 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 in most cases, and applicable

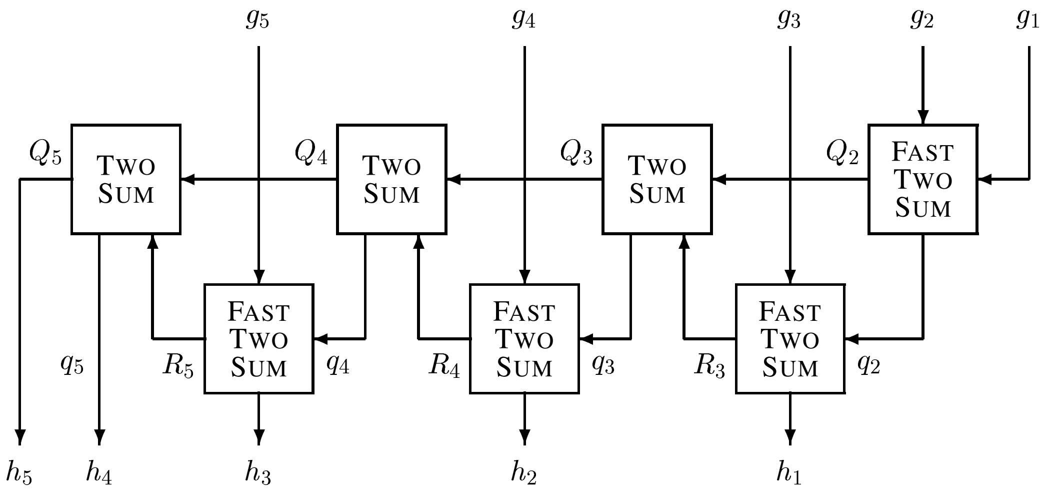

Figure 23: Operation of LINEAR-EXPANSION-SUM. $Q_i + q_i$ maintains an approximate running total. The FAST-TWO-SUM operations in the bottom row exist to clip a high-order bit off each q_i term, if necessary, before outputting it.

to single or double precision floating-point inputs. I invite working computational geometers to try my code in their implementations, and hope that it will save them from worrying about robustness so they may concentrate on geometry.

A Linear-Time Expansion Addition without Round-to-Even Tiebreaking

Theorem 24 *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 $p \geq 3$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. Then 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)
1    Merge e and f into a single sequence g, in order of
         nondecreasing magnitude (possibly with interspersed zeroes)
2    (Q_2, q_2) ⇐ FAST-TWO-SUM(g_2, g_1)
3    for i ⇐ 3 to m + n
4        (R_i, h_{i-2}) ⇐ FAST-TWO-SUM(g_i, q_{i-1})
5        (Q_i, q_i) ⇐ TWO-SUM(Q_{i-1}, R_i)
6    h_{m+n-1} ⇐ q_{m+n}
7    h_{m+n} ⇐ Q_{m+n}
8    return h
```

$Q_i + q_i$ is an approximate sum of the first i components of g; see Figure 23.

Proof: At the end of each iteration of the **for** loop, the invariant $Q_i + q_i + \sum_{j=1}^{i-2} h_j = \sum_{j=1}^{i} g_j$ holds. Certainly this invariant holds for $i = 2$ after Line 2 is executed. From Lines 4 and 5, we have that $Q_i + q_i +$

$h_{i-2} = Q_{i-1} + q_{i-1} + g_i$; the invariant follows by induction. (The use of FAST-TWO-SUM in Line 4 will be justified shortly.) This assures us that after Lines 6 and 7 are executed, $\sum_{j=1}^{m+n} h_j = \sum_{j=1}^{m+n} g_j$, so the algorithm produces a correct sum.

The proof that h is nonoverlapping and increasing relies on the fact that the components of g are summed in order from smallest to largest, so the running total $Q_i + q_i$ never grows much larger than the next component to be summed. Specifically, I prove by induction that the exponent of Q_i is at most one greater than the exponent of g_{i+1}, and the components $h_1, \ldots, h_{i-1}$ are nonoverlapping and in order of increasing magnitude (excepting zeros). This statement holds for $i = 2$ because $|Q_2| = |g_1 \oplus g_2| \leq 2|g_2| \leq 2|g_3|$. To prove the statement in the general case, assume (for the inductive hypothesis) that the exponent of Q_{i-1} is at most one greater than the exponent of g_i, and the components $h_1, \ldots, h_{i-2}$ are nonoverlapping and increasing.

q_{i-1} is the roundoff error of the TWO-SUM operation that produces Q_{i-1}, so $|q_{i-1}| \leq \frac{1}{2}\text{ulp}(Q_{i-1})$. This inequality and the inductive hypothesis imply that $|q_{i-1}| \leq \text{ulp}(g_i)$, which justifies the use of a FAST-TWO-SUM operation in Line 4. This operation produces the sum $|R_i + h_{i-2}| = |g_i + q_{i-1}| < (2^p + 1)\text{ulp}(g_i)$. Corollary 8(a) implies that $|h_{i-2}| < \text{ulp}(g_i)$. Because $h_1, \ldots, h_{i-2}$ are nonoverlapping, we have the bound $|\sum_{j=1}^{i-2} h_j| < \text{ulp}(g_i) \leq \text{ulp}(g_{i+1})$.

Assume without loss of generality that the exponent of g_{i+1} is $p - 1$, so that $\text{ulp}(g_{i+1}) = 1$, and $|g_1|, |g_2|, \ldots, |g_{i+1}|$ are bounded below 2^p. Because g is formed by merging two nonoverlapping increasing expansions, $|\sum_{j=1}^{i} g_j| < 2^p + 2^{p-1}$. Consider, for instance, if $g_{i+1} = 1000$ (in four-bit arithmetic); then $|\sum_{j=1}^{i} g_j|$ can be no greater than the sum of $1111.1111\ldots$ and $111.1111\ldots$.

Substituting these bounds into the invariant given at the beginning of this proof, we have $|Q_i + q_i| \leq |\sum_{j=1}^{i-2} h_j| + |\sum_{j=1}^{i} g_j| < 2^p + 2^{p-1} + 1$, which confirms that the exponent of Q_i is at most one greater than the exponent of g_{i+1}.

To show that h_{i-1} is larger than previous components of h (or is zero) and does not overlap them, observe from Figure 23 that h_{i-1} is formed (for $i \geq 3$) by summing g_{i+1}, R_i, and Q_{i-1}. It can be shown that all three of these are either equal to zero or too large to overlap h_{i-2}, and hence so is h_{i-1}. We have already seen that $|h_{i-2}| < \text{ulp}(g_i)$, which is bounded in turn by $\text{ulp}(g_{i+1})$. It is clear that $|h_{i-2}|$ is too small to overlap R_i because both are produced by a FAST-TWO-SUM operation. Finally, $|h_{i-2}|$ is too small to overlap Q_{i-1} because $|h_{i-2}| \leq |q_{i-1}|$ (applying Lemma 1 to Line 4), and $|q_{i-1}| \leq \frac{1}{2}\text{ulp}(Q_{i-1})$.

The foregoing discussion assumes that none of the input components is zero. If any of the g_i is zero, the corresponding output component h_{i-2} is also zero, and the accumulator values Q and q are unchanged ($Q_i = Q_{i-1}$, $q_i = q_{i-1}$). ■

B Why the Tiebreaking Rule is Important

Theorem 13 is complicated by the need to consider the tiebreaking rule. This appendix gives an example that proves that this complication is necessary to ensure that FAST-EXPANSION-SUM will produce nonoverlapping output. If one's processor does not use round-to-even tiebreaking, one might use instead an algorithm that is independent of the tiebreaking rule, such as the slower LINEAR-EXPANSION-SUM in Appendix A.

Section 2.4 gave examples that demonstrate that FAST-EXPANSION-SUM does not preserve the nonoverlapping or nonadjacent properties. The following example demonstrates that, in the absence of any assumption about the tiebreaking rule, FAST-EXPANSION-SUM does not preserve any property that implies the nonoverlapping property. (As we have seen, the round-to-even rule ensures that FAST-EXPANSION-SUM preserves the strongly nonoverlapping property.)

For simplicity, assume that four-bit arithmetic is used. Suppose the round-toward-zero rule is initially in effect. The incompressible expansions $2^{14} + 2^8 + 2^4 + 1$ and $2^{11} + 2^6 + 2^2$ can each be formed by summing their components with any expansion addition algorithm. Summing these two expansions, FAST-EXPANSION-SUM (with zero elimination) yields the expansion $1001 \times 2^{11} + 2^8 + 2^6 + 2^4 + 2^2 + 1$. Similarly, one can form the expansion $1001 \times 2^{10} + 2^7 + 2^5 + 2^3 + 2^1$. Summing these two in turn yields $1101 \times 2^{11} + 2^{10} + 1111 \times 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 1$, which is nonoverlapping but not strongly nonoverlapping.

Switching to the round-to-even rule, suppose FAST-EXPANSION-SUM is used to sum two copies of this expansion. The resulting "expansion" is $111 \times 2^{13} + -2^{11} + 2^{10} + -2^5 + 2^5 + -2^1$, which contains a pair of overlapping components. Hence, it is not safe to mix the round-toward-zero and round-to-even rules, and it is not possible to prove that FAST-EXPANSION-SUM produces nonoverlapping expansions for any tiebreaking rule.

Although the expansion above is not nonoverlapping, it is not particularly bad, in the sense that APPROXIMATE will nonetheless produce an accurate approximation of the expansion's value. It can be proven that, regardless of tiebreaking rule, FAST-EXPANSION-SUM preserves what I call the *weakly nonoverlapping* property, which allows only a small amount of overlap between components, easily fixed by compression. (Details are omitted here, but I am quite certain of the result. I produced a proof similar to that of Theorem 13, and rivalling it in complexity, before I discovered the strongly nonoverlapping property.) I conjecture that the geometric predicates of Section 4 work correctly regardless of tiebreaking rule.

References

[1] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic*. Algorithmica **17**(2):111–132, February 1997.

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

[3] C. Bradford Barber. *Computational Geometry with Imprecise Data and Arithmetic*. Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, October 1992. Available as Technical Report CS-TR-377-92.

[4] Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. *Exact Geometric Computation in LEDA*. Eleventh Annual Symposium on Computational Geometry (Vancouver, British Columbia, Canada), pages C18–C19. Association for Computing Machinery, June 1995.

[5] John Canny. *Some Algebraic and Geometric Computations in PSPACE*. 20th Annual Symposium on the Theory of Computing (Chicago, Illinois), pages 460–467. Association for Computing Machinery, May 1988.

[6] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation*. 33rd Annual Symposium on Foundations of Computer Science (Pittsburgh, Pennsylvania), pages 387–395. IEEE Computer Society Press, October 1992.

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

[8] Steven Fortune. *Stable Maintenance of Point Set Triangulations in Two Dimensions*. 30th Annual Symposium on Foundations of Computer Science, pages 494–499. IEEE Computer Society Press, 1989.

[9] ———. *Progress in Computational Geometry*. Directions in Geometric Computing (R. Martin, editor), chapter 3, pages 81–128. Information Geometers Ltd., 1993.

[10] ———. *Numerical Stability of Algorithms for 2D Delaunay Triangulations*. International Journal of Computational Geometry & Applications **5**(1–2):193–213, March–June 1995.

[11] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry*. Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 163–172. Association for Computing Machinery, May 1993.

[12] ———. *Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry*. ACM Transactions on Graphics **15**(3):223–248, July 1996.

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

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

[15] Christoph M. Hoffmann. *The Problems of Accuracy and Robustness in Geometric Computation*. Computer **22**(3):31–41, March 1989.

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

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

[18] Der-Tsai Lee and Bruce J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation*. International Journal of Computer and Information Sciences **9**(3):219–242, 1980.

[19] Seppo Linnainmaa. *Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums*. BIT **14**:167–202, 1974.

[20] Victor Milenkovic. *Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections using Rounded Arithmetic*. 30th Annual Symposium on Foundations of Computer Science, pages 500–505. IEEE Computer Society Press, 1989.

[21] N. E. Mnev. *The Universality Theorems on the Classification Problem of Configuration Varieties and Convex Polytopes Varieties*. Topology and Geometry - Rohlin Seminar (O. Ya. Viro, editor), Lecture Notes in Mathematics, volume 1346, pages 527–543. Springer-Verlag, 1988.

[22] Thomas Ottmann, Gerald Thiemt, and Christian Ullrich. *Numerical Stability of Geometric Algorithms*. Proceedings of the Third Annual Symposium on Computational Geometry, pages 119–125. Association for Computing Machinery, June 1987.

[23] Douglas M. Priest. *Algorithms for Arbitrary Precision Floating Point Arithmetic*. Tenth Symposium on Computer Arithmetic (Los Alamitos, California), pages 132–143. IEEE Computer Society Press, 1991.

[24] ______. *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, Berkeley, California, November 1992. Available by anonymous FTP to `ftp.icsi.berkeley.edu` as `pub/theory/priest-thesis.ps.Z`.

[25] Jonathan Richard Shewchuk. *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*. Applied Computational Geometry: Towards Geometric Engineering (Ming C. Lin and Dinesh Manocha, editors), Lecture Notes in Computer Science, volume 1148, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[26] Pat H. Sterbenz. *Floating-Point Computatation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[27] David F. Watson. *Computing the n-dimensional Delaunay Tessellation with Application to Voronoï Polytopes*. Computer Journal **24**(2):167–172, 1981.

[28] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.