Efficient Exact Arithmetic for Computational Geometry

Steven Fortune
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Christopher J. Van Wyk
Dept. of Mathematics and Computer Science
Drew University
Madison, New Jersey 07940

December 14, 1993

Abstract

We experiment with exact integer arithmetic to implement primitives for geometric algorithms. Naive use of exact arithmetic—either modular or multiprecision integer—increases execution time dramatically over the use of floating-point arithmetic. By combining tuned multiprecision integer arithmetic and a floating-point filter based on interval analysis, we can obtain the effect of exact integer arithmetic at a cost close to that of floating-point arithmetic. We describe an experimental expression compiler that conveniently packages our techniques.

1 Introduction

Geometric algorithms are usually described using exact arithmetic on real numbers (the "real RAM" model). Since no computer provides exact real arithmetic, programmers implementing geometric algorithms must find some substitution. Floating-point arithmetic is a common and convenient substitution, but there are no simple techniques that guarantee the reliability of the resulting program[8]. A few algorithms have been analyzed for stability when their arithmetic operations are carried out in floating-point[5, 12, 16, 17], but such analysis is difficult and algorithm-specific, and a general theory of stability is a distant goal.

Exact arithmetic is another possible substitution. Integer arithmetic suffices for many geometric algorithms: the use of homogeneous integer coordinates obviates rational numbers, and more general symbolic or algebraic numbers are rarely necessary. The difficulty is that the required integer precision exceeds the native precision of typical computers. Most geometric primitives involve evaluating the sign of a polynomial, and evaluating a polynomial of degree d over b-bit numbers requires intermediate values of bit-length about bd. Software multiprecision integer arithmetic seems rarely to be used in practice; perhaps programmers do not consider it at all, or perhaps they judge it too expensive or too inconvenient to use.

We report here on experiments with multiprecision integer arithmetic. We experimented with Delaunay triangulation algorithms in two and three dimensions. We had available several algorithms implemented using floating-point arithmetic; we replaced the floating-point primitives with integer-arithmetic primitives, implementing multiprecision arithmetic in various ways. The required primitives—the orientation test and the incircle test—are relatively simple, with maximum degree of 5 in three dimensions. Both primitives can be computed by evaluating the sign of a determinant; as noted in Section 2.1, evaluating a determinant over the integers has different complexity and requires different algorithms than over the reals.

For these implementations, between 20% and 50% of the total runtime is spent in floating-point geometric primitives. Depending on input bit-length and the choice of arithmetic, the integer primitives are between 40 and 140 times slower than corresponding floating-point primitives. Clearly, straightforward use of integer arithmetic implies at least an order of magnitude performance penalty. This large penalty is much more, for example, than the observed differences in performance among various 2-d Delaunay triangulation algorithms[6]. The data indicates that performance depends strongly on the bit-length of arithmetic, so the penalty for algorithms that involve more complex primitives is likely to be even worse. Of course, this comparison between floating-point and exact arithmetic is not completely fair, since a floating-point implementation is not always correct; nevertheless, the imperfect floating-point Delaunay triangulation implementations that we

0

used are still reliable enough to be useful.

We present two techniques that dramatically improve the efficiency of multiprecision integer arithmetic. The first is to tune fixed arithmetic expressions carefully. This technique is most useful if most operations involve integers of relatively short bit-length; in such cases we observed a performance improvement of between 4 and 10 over other multiprecision methods. The second technique is to use floating-point arithmetic as a filter to cull out easy cases. The efficacy of the filter depends heavily upon the details of the filter and the distribution of the input.

In order to be a useful tool for implementing geometric algorithms, multiprecision integer arithmetic must not only be efficient: it must also have a convenient interface. We tried several variants of our techniques that could be implemented using standard programming languages, but were unable to obtain acceptable performance. For best performance, our techniques require static analysis of the arithmetic expressions that define geometric predicates. Hence we developed an "expression compiler" that conveniently incorporates our techniques. This expression compiler is still under development; we report below on its current status.

Other work. There is little literature on the use of exact arithmetic in computational geometry. Karasick *et al.*[9] studied the use of exact rational (not integer) arithmetic: naive use of rational arithmetic in the 2-d divide-and-conquer Delaunay triangulation algorithm cost a performance penalty of 10^4 over floating-point. Extensive engineering, including adaptive-precision arithmetic and changes to the algorithm, reduced the overall performance penalty to a factor of about 4. The filtering technique that we use is abstractly similar to their adaptive precision, but the engineering and resulting performance are quite different.

2 Experimental results

We experimented with three implementations of Delaunay triangulation algorithms, the 2-d flipping algorithm[11], the 2-d divide-and-conquer algorithm[7, 15], and a random-incremental convex hull algorithm [4] used to compute Delaunay triangulations via the lifting map[7]. The last implementation handles general dimension, but we used it only for three-dimensional Delaunay triangulations, corresponding to four-dimensional convex hulls. All programs are written in the C language.

All the algorithms require the orientation test and the incircle test. The orientation test on d+1 points in R^d can be expressed as the sign of the determinant

$$\begin{vmatrix} p_1 & p_2 & p_3 & 1 \\ q_1 & q_2 & q_3 & 1 \\ r_1 & r_2 & r_3 & 1 \\ s_1 & s_2 & s_3 & 1 \end{vmatrix}$$

Figure 1: 3-d orientation test on points p, q, r, s.

of a $(d{+}1) \times (d{+}1)$ matrix whose last column is all ones (see Figure 1). The last column can easily be eliminated, leaving a $d \times d$ matrix with integer entries of about the same bit-length as the point coordinates. The incircle test on d+2 points in R^d can be reduced to the orientation test in R^{d+1} using the lifting map $\lambda(p_1, \ldots, p_d) = (p_1, \ldots, p_d, p_1^2 + \cdots + p_d^2)$. Eliminating the last column yields a $(d{+}1) \times (d{+}1)$ matrix with integer entries of about coordinate bit-length, except in one column whose entries are twice as long.

The random-incremental convex hull implementation replaces the orientation test on d+1 points by the computation of the hyperplane through the first d points, followed by a dot product of the remaining point with the hyperplane normal. Future orientation tests with the same first d points can be replaced by a dot product. The hyperplane could be computed at about the same cost as the orientation test; to improve numerical stability the implementation instead uses Gaussian elimination with full pivoting, and performs the first elimination step using a minimum spanning tree technique[5].

Figure 2 gives statistics on the implementations' performance. The timings are for a VAX 8550. Timings in this paper were obtained in three ways: by timing the entire program, by profiling, and by a "statement timing utility"[2]. Timings by the various techniques typically vary by at most 10%.

2.1 Evaluating Determinants

Figure 3 shows the multiplication complexity of four different methods of determinant evaluation (*cf.* Figure 4, which shows multiplication complexity using real arithmetic). We use a 'digit' model: multiplying two one-digit numbers costs one; multiplying a k-digit number by an l-digit number costs kl. (Digit length is defined by the computer architecture; typical 32-bit integer arithmetic yields a digit length of 15 or 16 bits. Asymptotically faster multiplication algorithms are irrelevant for small d.) We also assume that matrix entries are small enough that the determinant of any $k \times k$ submatrix requires only k digits. (By Hadamard's inequality, the absolute value of the determinant of a $d \times d$ matrix with b-bit entries is at most $d!2^{bd}$; hence it suffices to assume that the digits are at least $b + \log d$ bits long.) Roughly

Algorithm	d	n	time (sec)	detailed time accounting
flipping	2	5000	10.2	22 % in 2-d orientation (212000 tests) 22 % in 2-d incircle (65000 tests)
divide-and-conquer	2	5000	8.5	8 % in 2-d orientation (78000 tests) 10 % in 2-d incircle (34000 tests)
random-incremental convex hull	3	1000	18.8	29 % in Gaussian elim. (25000 times) 10 % in Gaussian elim. heuristics 9 % in dot product (100000 times)

Figure 2: Delaunay triangulation timings: input points are chosen pseudouniformly in the unit cube.

speaking, multiplying the length of every matrix entry by k causes the number of multiplications for modular arithmetic to grow by k and for cofactor expansion and dynamic programming by k^2.

The figures refer to the following methods of evaluating determinants:

1. Cofactor expansion. This requires d subdeterminants followed by d multiplications. The recurrence for real arithmetic is $C(d) = dC(d-1) + d$; the recurrence for the single-digit model is $C(d) = dC(d-1) + d(d-1)$.

2. Dynamic programming. Evaluate all $\binom{d}{2}$ determinants of all 2×2 minors of the first two rows, then the $\binom{d}{3}$ determinants of all 3×3 minors of the first three rows, and so forth, until the entire determinant is computed. (This is cofactor expansion taking advantage of common subexpressions.) The complexity in the real-arithmetic model is $\sum_{i=1}^{d} \binom{d}{i} i = d(2^{d-1}-1)$; in the single-digit model it is $\sum_{i=1}^{d} \binom{d}{i} i(i-1) = d(d-1)2^{d-2}$.

3. One-step elimination[1]. Gaussian elimination introduces zeroes below the main diagonal by applying elementary transformations to the matrix; the usual transformation computes the reciprocal of the pivot element. "Cross multiplication" allows one to eliminate the division. Implemented naively, cross multiplication doubles the bit-length of matrix entries every time a column is annihilated. Bareiss[1] attributes to Camille Jordan the observation that the pivot element of the previous iteration divides every new entry resulting from the cross multiplication. Using an algorithm that replaces the new entry by the quotient, Bareiss obtains an algorithm with multiplication complexity $O(d^5)$. For small d the algorithm is always worse than dynamic programming, so it is not included in Figure 3.

4. Modular arithmetic. Choose a set of primes such that the value of the determinant is guaranteed to be less than the product of the primes. Evaluate the determinant modulo each of the primes independently, then use the Chinese remainder theorem to reconstruct the determinant. The complexity depends upon the determinant evaluation strategy. Gaussian elimination requires modular inverse, which is expensive, though for small enough primes it is plausible to do modular inverse by table lookup. Dynamic programming does not require inverse, but requires more operations. We obtained the table entries in Figure 3 by multiplying the corresponding entries in Figure 4 by $2d$: we charge two for each modular operation (a multiplication and a division) and we assume that exactly d primes suffice; this does not include the cost of modular inverse or determinant reconstruction.

Clarkson[3] has recently given an algorithm to compute the sign of a determinant; for a $d \times d$ matrix with b-bit entries, it requires $O(d^3)$ arithmetic operations on numbers of bit-length $O(b+d)$. We do not know how to give a fair operation count for his algorithm: it is adapative and it apparently requires (expensive) square roots. In the best case, Clarkson's algorithm performs a Gram-Schmidt orthogonalization (d^3 operations) followed by Gaussian elimination ($d^3/3$ operations), for a total of about four times the cost of Gaussian elimination; the precise worst-case operation count is unclear.

2.2 Naive Exact Arithmetic

Of the many existing multiprecision arithmetic packages, we chose two: "mp", a multiprecision integer arithmetic package that comes with our Unix system, and "BigNum"[14], distributed from INRIA. The mp package is written entirely in C. BigNum has a C interface to VAX assembly language routines that implement the actual arithmetic operations. We used the high-level interface ("Bz") to BigNum; BigNum has a low-level interface ("Bn") that may be more efficient, but is also significantly more complicated.

We wrote a modular arithmetic package in C++; operator overloading provides a convenient interface. Our

Method	bound	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
cofactor expansion	$\sim d!$	12	60	320	1950	13692
dynamic programming	$d(d-1)2^{d-2}$	12	48	160	480	1344
modular arithmetic (DP)	$\sim d^2 2^d$	54	224	750	2232	6174
modular arithmetic (GE)	$\sim 2d^4/3$	60	178	440	900	1652

Figure 3: Estimated single-digit multiplication complexity of determinant evaluation, digit model

Method	bound	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
cofactor expansion	$\sim d!$	9	40	205	1236	8659
dynamic programming	$d(2^{d-1}-1)$	9	28	75	186	441
Gaussian elimination	$\sim d^3/3$	10	23	44	75	118

Figure 4: Multiplication complexity of determinant evaluation, real arithmetic

Test	time (μsec)
2-d incircle	30
3-d incircle	75

Figure 5: Floating-point timings, dyn. prog.

Method	$b = 20$	$b = 30$	$b = 40$	$b = 50$
mp	1800	2300	2800	3500
BigNum	2500	2500	2700	2850
mod arith	1200	1800	2400	3100

Figure 6: 2-d incircle times, dyn. prog. (μsec)

Method	$b = 20$	$b = 30$	$b = 40$	$b = 50$
mp	5200	6800	8100	10500
mod arith	3400	5100	6600	8000

Figure 7: 3-d incircle times, dyn. prog. (μsec)

program uses primes that fit into 15 bits, since the product of two primes must fit into a signed[1] 32-bit integer word. Because the native signed integer modulus operation is expensive we avoided using it whenever possible, as suggested by Knuth[10]. To reduce the overhead of creating and destroying temporaries, we implemented modular assignment operators ("+=" and "×=") instead of the more natural binary operators.

Figures 6 and 7 give execution times of the various kinds of incircle tests as a function of the bit-length of the point coordinates. For reference, Figure 5 gives execution time in double-precision floating-point arithmetic. In each case we used dynamic programming to implement the incircle test. When the mp version of the incircle test was substituted into the implementation of the 2-d flipping algorithm, and the program was run on 5000 input points chosen with pseudorandom 24-bit integer coordinates, the new runtime was about 136 seconds, more than 13 times the floating-point runtime. Though we did not substitute an exact 3-d incircle test into the convex hull implementation, we can estimate a lower bound on its runtime as follows. Assume input points with 30-bit integer coordinates. The time for a hyperplane computation is about the same as for a 3-d incircle test, or about 6800μs. The computation of 25000 hyperplanes would require about 170 seconds. This would give a total runtime at least ten times that of the floating-point implementation.

The poor performance of the multiprecision arithmetic packages is surprising: the operation counts predicted neither the magnitude of the increase in runtimes nor the relative order for different versions of exact arithmetic. The source code for the multiprecision arithmetic packages reveals that they incur enormous overhead on arithmetic operations. For example, to perform a single addition, the BigNum package calls about 10 internal subroutines, including one for storage allocation; the subroutines that contain loops are written in assembler. The mp package uses fewer subroutine calls, but the inner loops are written in C. (The two implementation strategies explain the crossover in runtimes in Figure 6; indeed, when we tried the 2-d incircle test with 200-bit inputs, the BigNum version was four times faster than mp.) We did not know how to improve the performance of modular arithmetic substantially, so we decided to try to improve multiprecision integer arithmetic.

[1]Unsigned integer modulus in C is performed by subroutine call on our VAX, so its cost is prohibitive.

2.3 Tuning Exact Arithmetic

While there is no obviously better implementation strategy for general-purpose multiprecision arithmetic, computational geometry imposes special constraints on the use of multiprecision arithmetic. First, it is reasonable to assume that most arithmetic operations are of "medium length," far shorter than one hundred machine words; we must reduce overhead per operation to get good performance. Second, while the arithmetic expressions used in computational geometry are fairly complicated, they do not change dynamically; it is reasonable to optimize the evaluation of an entire expression, rather than just a single operation.

To see how well we could tune multiprecision integer arithmetic, we wrote a (throwaway) expression compiler: the input is a multivariable expression over $\{+, -, \times\}$ and a specification of the bit-lengths of the variables. The output is a C program that evaluates the expression exactly and then returns the sign of the value as a C variable. The expression compiler estimates the bit-length of each intermediate value and uses the estimate to generate straight-line C code for each operation.

We represent a multiprecision integer value a as the sum of double-precision floating-point variables

$$a_0 + \cdots + a_m, \text{ where } a_i = a_i' \cdot 2^{ri},\ a_i' \text{ integral}, \qquad (1)$$

and r is the digit (or radix) length. We use double-precision variables since they allow machine operations on values around 53 bits long, whereas the integer machine operations work on 32-bit values.[2] IEEE floating-point arithmetic guarantees that an arithmetic operation is exact as long as the result is exactly representable as a floating-point value.[3]

We say that a multiprecision value such as (1) is *normalized* if each a_i' is at most 2^r in absolute value. Since normalization is expensive, we do not require that multiprecision values be normalized; hence the representation of a multiprecision value is not unique.

The multiprecision sum $c := a + b$, where $a = a_0 + \cdots + a_m$ and $b = b_0 + \cdots + b_m$, can be implemented simply as the floating-point sums

$$c_0 := a_0 + b_0, c_1 := a_1 + b_1, \ldots, c_m := a_m + b_m, \qquad (2)$$

as long as each result is exactly representable. The multiprecision product $c := ab$, where $a = a_0 + \cdots + a_m$ and $b = b_0 + \cdots + b_n$, is implemented using the usual $O(mn)$ algorithm, thus

$$c_k := \sum_j a_j b_{k-j}, \qquad (3)$$

again as long as each result is exactly representable.

Often with (3) and sometimes with (2), some term in the result is not exactly representable as a floating-point value, and the operands must be normalized before the operation. In order of increasing i we execute the simultaneous assignment

$$a_i, a_{i+1} := a_i \bmod 2^{ri}, a_{i+1} + a_i - (a_i \bmod 2^{ri}),$$

which guarantees that for each i, a_i' is at most 2^r in absolute value. Instead of computing the mod using an expensive library subroutine, we add a large constant to guarantee aligned bit positions in the normalized floating-point significands, perform the modulus operation by converting to single-precision floating-point and subtracting, and finish by subtracting the large constant. (A less portable bitwise "and" on the significands could also be used to perform the modulus operation.)

The choice of the radix bitlength r is subtle. To minimize the number of variables needed to represent a value, it should be as large as possible, *viz.* about half the bitlength of a double-precision significand. Since determinant evaluation typically alternates sums and products, however, and we would like to be able to add several products without normalization, we might want to reduce the radix length slightly. The results reported here are for $r = 23$.

Figure 12 shows the code generated by a later version of the expression compiler. The code is not easy to understand, nor would it be easy to write by hand. Figures 8 and 9 give statistics on incircle tests in two and three dimensions generated by this method. For reference, the first column gives the operation count for exact evaluation in 53-bit floating point, assuming bit-length is restricted not to need multiple precision. The runtimes are much faster than for the other multiprecision methods, though still much slower than floating-point arithmetic. The change in runtime is due to the use of double-precision floating-point, the lazy normalization strategy, and the elimination of overhead for loops, subroutines, and memory management. We do not know the individual contribution of each change, or even if every change is beneficial; further tuning could well reduce the runtime further.

2.4 A Floating-Point Filter

Another way to improve overall performance is to reduce the use of exact multiprecision arithmetic. An obvious

[2] Floating-point arithmetic on modern workstations has been heavily optimized: it is hard to measure meaningfully, in part because of pipelining, but it appears that 53-bit double-precision floating-point multiplication on the SGI R3000 is four times faster than 32-bit integer multiplication. It would be a shame not to take advantage of such engineering.

[3] While the VAX does not have IEEE arithmetic, it is close enough for similar techniques to work.

	$b = 11$	$b = 20$	$b = 30$	$b = 40$	$b = 50$
Additions	18	30	124	145	290
Multiplications	17	29	98	116	225
Normalizations	0	9	41	50	90
Total time (μsec)	30	91	323	409	760

Figure 8: 2-d incircle test statistics, dynamic programming, tuned arithmetic

	$b = 8$	$b = 20$	$b = 30$	$b = 40$	$b = 50$
Additions	41	86	311	383	733
Multiplications	40	80	254	319	598
Normalizations	0	27	92	115	198
Total time (μsec)	75	260	791	1020	1820

Figure 9: 3-d incircle test statistics, dynamic programming, tuned arithmetic

technique to try is interval analysis: replace the evaluation of an expression by the evaluation of guaranteed upper and lower bounds on its value. When the sign of the expression is needed, if the bounding interval does not contain zero, then the sign is known; otherwise, the expression needs to be reevaluated at higher precision.

Rather than conventional interval analysis, we implemented the following somewhat coarser technique. Given an arithmetic expression to be evaluated over integers of bounded length, we can estimate bounds on each of the intermediate values in the expression. If the expression is evaluated using floating-point arithmetic, we can also inductively estimate the maximum absolute error in each intermediate value. For example, the absolute error in the floating-point sum $a \oplus b$ is at most the sum of the absolute errors of a and b and the error in the floating-point $\oplus$ itself. The error in $\oplus$ can be at most in the low-order bit, so it is at most 2^{l-f+1}, where 2^l is the estimate of the magnitude of $a + b$ and f is the number of bits in the floating-point significand. A similar computation is possible for floating-point multiplication. When the sign of the expression is needed, we compare the absolute value of the floating-point result with the error estimate, and resort to exact arithmetic only if it is smaller. The advantage of this approach is that the uncertainty test is cheap—a single comparison—since the error estimate can be determined statically; the disadvantage is that the error estimate is pessimistic.

We tried this approach with the flipping algorithm in two dimensions and the convex hull algorithm in three dimensions. Figure 10 shows how often the double-precision floating-point evaluation failed for various distributions. The "random" distribution is for points chosen with pseudouniform 24-bit coefficients. "Cocircular" points are chosen from a set $\{(a, b) : a^2 + b^2 = c\}$; it is hard to find a constant c so that the set is large. The "approximately cocircular" distribution is a set of 5000 points chosen pseudorandomly on a circle of radius 2^{24} using floating-point, then rounding to integer coordinates. A "partial lattice" is a set of points chosen pseudorandomly from a square or cubical lattice with some probability p between .1 and 1; the size of the point set varies between 3000 and 10000. The various non-random point distributions simulate input data that is not in general position, a case that is encountered often in practice.

Distribution	d	Exact computations
random	2	0 out of 10000
cocircular	2	all
approx. cocircular	2	$\sim$ 50%
partial lattice	2	5-10 %
random	3	1-3 out of 100000
partial lattice	3	4-21 %

Figure 10: Frequency of incircle tests requiring exact arithmetic

3 Packaging

The techniques in Sections 2.3 and 2.4 have the potential to improve the performance of exact arithmetic substantially. We turned our attention to packaging the techniques so that it would be easy to develop geometric primitives. We knew that implementation of geometric algorithms is particularly challenging; making reliable primitives available would let programmers concentrate on other implementation problems. Thus, a principal goal was to devise a convenient interface. We would be willing to compromise somewhat on performance to simplify the interface, though not to the extent required by existing multiprecision arithmetic packages.

3.1 Lazy Evaluation

One approach is lazy evaluation of arithmetic expressions. Instead of evaluating an expression exactly, the expression should be evaluated in floating-point, while accumulating error bounds and enough state so that, if necessary, the expression can be reevaluated exactly later. We planned to provide a C++ class *number*, and to overload the arithmetic operations to accumulate error bounds and save state behind the scenes.

A *number* object stores members *value*, the floating-point value, *error*, the error bound, and *op*, which records the operation and operands that produced the value. The collection of *op* fields defines an arithmetic expression graph. The operation $c := a \times b$ allocates memory for c and then executes

$$
\begin{aligned}
value(c) &:= value(a) \times value(b) \\
error(c) &:= |value(a)| \times error(b) \\
&\quad + |value(b)| \times error(a) \\
&\quad + |value(c)| \\
op(c) &:= (\times, a, b)
\end{aligned}
$$

A comparison $c > 0$ is implemented by

if $|value(c)| > \epsilon \times error(c)$
then if $value(c) > 0$ then true else false
else $exacteval(c) > 0$

where *exacteval* reevaluates its argument exactly, using the arithmetic expression graph saved in *op*.

We implemented a skeleton version of this approach, omitting the *exacteval* backend. To handle storage management, we used a library memory allocator tuned to the appropriate record size; we also maintained reference counts so that unreferenced records could be reclaimed. After some tuning, each overloaded operation was about 70 times as expensive as the corresponding primitive floating-point operation. Evidently the per-operation overhead required for dynamic state-saving is not so different from the per-operation overhead of exact arithmetic. We did not pursue this approach further.

3.2 Dynamic error bounds

Another approach would be to compute error bounds and not save state. We implemented a C++ class *number* with arithmetic operators overloaded to maintain just the *value* and *error* members described above. This approach avoids allocating memory with every arithmetic operation, since the *numbers* can be allocated statically or on the runtime stack. The 2-d incircle test requires about 480 μsec; in other words, the per-operation overhead is a factor of about 16. Although this overhead seems high, it is not out of line with predictions based on raw operation counts or conventional estimates of the cost of interval analysis.

In principle we could implement geometric primitives using this version of class *number*. A geometric primitive could be coded in straightforward fashion, using overloaded arithmetic operations. A comparison would have three outcomes, 'true', 'false' and 'uncertain'; in the 'uncertain' case the primitive would have to redo its calculation in multiprecision arithmetic. For a single implementation, this approach might be reasonable, since only standard components are needed. As a general solution, however, it is not attractive. If floating-point geometric primitives contribute 20% of the runtime, computing the error bounds alone slows the whole implementation by a factor of four. The programmer will still need to handle 'uncertain' outcomes, possibly by resorting to multiprecision arithmetic. We did not pursue this approach either.

The cost of dynamic error bounds can be eliminated if the error bounds are computed statically, for example using the approach in Section 2.4. The error bounds could be computed by hand, but this is tedious and error-prone. The error bounds can be computed automatically; this requires that the whole arithmetic expression be available for analysis. Of course, once the whole expression is available, carefully tuned multiprecision arithmetic can be generated as well. We decided to see if we could make the expression compiler described in Section 2.3 into a useful tool. The next section describes the result.

4 An Expression Compiler

The expression compiler is a precompiler: its input is a set of arithmetic expressions over the integers, each of which defines a geometric predicate; its output is a set of C++ functions that evaluate the expressions.

The expression language includes variables, constants, arithmetic operations $+$, $-$, $\times$, assignment, conditionals, and a few predefined functions such as *sign* (which returns -1, 0, or $+1$). To simplify coding one can define 'tuples' of fixed arity, and use 'macros' to substitute parameters. Figure 11 gives a fairly elaborate definition of the 2-d incircle test using the composition of the lifting map with a reduction of 3-d orientation to homogeneous

2-d orientation. (The simpler straight-line version of 'Incircle2d' would not have illustrated as many features of the expression compiler.) Figure 12 shows a simplified version of part of the output generated from Figure 11; the actual output is about 180 lines of C++.

The expression language itself denotes expressions with integers of arbitrary size. However, each generated C++ function is specialized to work on integers of fixed maximum size. A 'struct' facility specifies the bit-length of input parameters and how they are represented in the calling C++ program (for example, as an 'int' or a 'double'). The same expression could be used to generate several C++ functions, each for arguments of different bit-lengths. For a particular function, the expression compiler uses the bit-length specification to estimate the bit-length of intermediate values and error bounds on floating-point approximations, as described in Sections 2.3 and 2.4.

A generated C++ function is guaranteed to return the result defined by arbitrary-precision integer arithmetic. The generated code uses floating-point arithmetic as a filter. For example, if the result is the sign of an arithmetic expression, then the expression is evaluated in floating-point arithmetic first, and evaluates the expression exactly only if the magnitude of the result is less than the statically-determined error bound. This lazy evaluation strategy is extended to arbitrary Boolean tests; thus a Boolean test in the expression language usually generates two tests in the C++ program, one that determines if the floating-point computation is reliable and the other for the test itself. The expression compiler generates code that avoids evaluating an expression more than once. Since the generated code has more tests than appear in the source language, and since the dynamic flow of control cannot be predicted statically, the actual code generation algorithm is rather complex. We omit details of the algorithm here.

The expression language does not include division: we do not know a convincing example that requires division, and division substantially complicates the static determination of bit-lengths and error bounds. The expression language also does not include any mechanism for tuples of variable-arity; this implies, for example, that the orientation test must be coded separately in each dimension.

The current version of the expression compiler is about 2400 lines of C++. The time required to generate C++ code for examples like 'Incircle2d' is negligible, perhaps 5% of the time subsequently required by the C++ compiler. Substituting the generated 2-d incircle test into the floating-point flipping algorithm changes its performance in a way that depends upon the input distribution: for random inputs with 24-bit coordinates, runtime actually decreased slightly; for approximately cocircular points, where about 50% of the incircle tests require exact computation, runtime increased by about 65%.

4.1 Possible extensions

The current expression compiler produces C++ procedures whose return value is only a 32-bit integer. This is perfectly adequate for predicates such as the orientation test, where only the sign of a multiprecision value is required. Some applications, however, seem to require a function to return a multiprecision value.

Consider an algorithm in which the orientation test in R^d is to be performed repeatedly with d points fixed and only one point varying; the random-incremental convex hull algorithm is an example. Then it is desirable to compute the coefficients of the hyperplane through the d points, so that a subsequent orientation test becomes a dot product. However, the hyperplane coefficients have bit-length about $d-1$ times the bit-length of the point coordinates, and are expensive to compute exactly. One can imagine various evaluation strategies: (1) do all computations exactly; (2) evaluate the hyperplane coefficients in floating-point and perform the dot products in floating-point, using exact arithmetic only if the floating-point answer is uncertain; (3) like (2), except compute the hyperplane coefficients exactly, then round to floating-point; (4) delay evaluation of the hyperplane coefficient until a dot product is actually needed, then use (2) or (3). Strategy (3) might be better than (2) because the error bound in the dot product is smaller, so fewer exact dot products might be required.

Strategy (1) is easiest to implement but also the most expensive. Once one of strategies (2)-(4) is available, it is easy for the expression compiler to provide the other two. In each case the expression compiler must be able to associate the computation of the hyperplane coefficients with their use in the dot-product computation, in order to estimate bit-lengths and error bounds statically. In addition, strategies (2) and (4) require that the dot-product code have access to the arguments to the hyperplane computation. This implies either copying, pointers, or user assistance; the linguistic challenge is to provide a convenient interface.

In some circumstances a computed value is desired, but need not be exact. Victor Milenkovic suggested adding "guaranteed rounding": the statement

$$a :\approx b \ni p(a,b);$$

means "assign to a an approximation to b that satisfies predicate $p(a,b)$." Predicate p might be an absolute or relative error bound; in general p can be the sign of an arbitrary arithmetic expression. The implementation of the assignment is to assign to a the floating-point approximation to b if predicate p is satisfied, otherwise to assign to b the exact value of a. In many cases the truth of p can be determined using just the floating-point approximation to b and statically-determined error bounds on b, without knowing the exact value of b. This feature might be useful for graphics output, where point coordinates are needed with a particular accuracy; exact computation of coordinates can be avoided unless their computation is ill-conditioned.

5 Conclusions

The expression compiler has proven useful for producing reliable implementations of some geometric algorithms. Beyond Delaunay triangulation algorithms, we have used the expression compiler for a program that clips a line to a polygon, a conceptually simple problem that is complicated in detail. While exact arithmetic does not simplify the analysis of various degenerate cases, it is reassuring to know that primitives give mathematically exact answers. We have confidence that our implementation reliably handles all inputs.

We do not know the class of geometric algorithms for which exact arithmetic is reasonable, even with the expression compiler. The salient factor is the complexity of the algebraic expressions that define the geometric primitives. It is easy to find natural geometric predicates of algebraic degree 10 or more; an example is the 2-d Incircle test on line segments. Cascading leads to rapid degree growth; for example, the coordinates of the point of intersection of two line segments have about four times the bit-length of the coordinates of the segment endpoints. Since the cost of exact arithmetic increases with the algebraic degree of the expression and the efficacy of the floating-point filter decreases with the nesting depth of the expression, the algebraic complexity of primitives is still a concern.

References

[1] E. H. Bareiss, Computational solution of matrix problems over an integral domain, *J. Inst. Maths Applics* 10:68–104, 1972.

[2] J. L. Bentley, B. Kernighan, C. Van Wyk, An elementary C cost model, *UNIX Review* 9(2):38–48, 1991.

[3] K.L. Clarkson, Safe and effective determinant evaluation, *33th Symp. on Found. Comp. Sci.* 387–395, 1992.

[4] K.L. Clarkson, K. Mehlhorn, R. Seidel, Four results on randomized incremental constructions, *Symp. Theor. Aspects of Comp. Sci.*, 1992.

[5] S. J. Fortune, Numerical stability of algorithms for Delaunay triangulations, *Proc. Eighth Ann. Symp. Comp. Geom* 83–92, 1992.

[6] S. J. Fortune, Voronoi diagrams and Delaunay triangulations, *Euclidean Geometry and Computers*, World Scientific Publishing Co., D.A. Du, F.K. Hwang, eds., 1992.

[7] L.J. Guibas, J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Trans. Graphics* 4(2):74–123, 1985.

[8] C. Hoffmann, *Geometric and Solid Modeling*, Morgan Kaufmann Publishers, 1989.

[9] M. Karasick, D. Lieber, L. R. Nackman, Efficient Delaunay triangulation using rational arithmetic, *ACM Trans. Graphics* **10**(1):71–91, 1990.

[10] D. E. Knuth, *Seminumerical Algorithms*, Volume 2 of *The Art of Computer Programming*, 2d ed., Addison-Wesley, 1981.

[11] C.L. Lawson, Software for C^1 surface interpolation, *Mathematical Software III* 161–194, J.R. Rice, ed., Academic Press, 1977.

[12] Victor Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[13] R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM, 1979.

[14] B. Serpette, J. Vuillemin, J.C. Hervé. BigNum: a portable and efficient package for arbitrary-precision arithmetic, INRIA.

[15] M. Shamos, D. Hoey, Closest-point problems, *Proc. 16th Ann. Symp. Found. Comp. Sci.* 151–162, 1975.

[16] K. Sugihara, M. Iri, Geometric algorithms in finite-precision arithmetic, Research Memorandum RMI 88–10, University of Tokyo, September, 1988.

[17] K. Sugihara, M. Iri, Construction of the Voronoi diagram for one million generators in single precision arithmetic, First Canadian Conference on Computational Geometry, Montreal, Canada, 1989.

```
struct integer = int<31>;
tuple Point2d(x,y), HPoint2d(x,y,w), Point3d(x,y,z);
macro HOrient2d(p1:HPoint2d, p2:HPoint2d, p3:HPoint2d) =
        sign (p1.x*(p2.y*p3.w - p2.w*p3.y)
           - p2.x*(p1.y*p3.w - p1.w*p3.y)
           + p3.x*(p1.y*p2.w - p1.w*p2.y));
macro Diff3d(p:Point3d, q:Point3d) =
        HPoint2d(p.x-q.x, p.y-q.y, p.z-q.z);
macro Orient3d(p1:Point3d, p2:Point3d, p3:Point3d, p4:Point3d) =
        HOrient2d(Diff3d(p2,p1), Diff3d(p3,p1), Diff3d(p4,p1));
macro Lift2d(p:Point2d) =
        Point3d(p.x, p.y, p.x*p.x + p.y*p.y);
macro Incircle2d(p1:Point2d, p2:Point2d, p3:Point2d, p4:Point2d) =
        Orient3d(Lift2d(p1), Lift2d(p2), Lift2d(p3), Lift2d(p4));
proc Incircle2d;
```

Figure 11: The 2d Incircle predicate

```
typedef int integer;
typedef struct {
   integer x;
   integer y;
} Point2d;

int Incircle2d(Point2d& p1, Point2d& p2, Point2d& p3, Point2d& p4){
double t1 = p3.y-p1.y;
double t2 = p1.x*p1.x+p1.y*p1.y;
double t3 = p4.x*p4.x+p4.y*p4.y-t2;
double t4 = p3.x*p3.x+p3.y*p3.y-t2;
double t5 = p4.y-p1.y;
double t6 = p2.y-p1.y;
double t7 = p2.x*p2.x+p2.y*p2.y-t2;
double t8 = (p2.x-p1.x)*(t1*t3-t4*t5)-(p3.x-p1.x)*(t6*t3-t7*t5)+(p4.x-p1.x)*(t6*t4-t7*t1);
if (-P2[86]<t8 && t8<P2[86]) {
  double t10 = float(P2x3[45]+p1.x)-P2x3[45];
  double t9 = p1.x-t10; /* t9 + t10 = p1.x */
  double t12 = float(P2x3[45]+p1.y)-P2x3[45];
  double t11 = p1.y-t12; /* t11 + t12 = p1.y */
  double t13 = t9*t9+t11*t11;
  double t14 = t9*t12+t12*t9+t10*t11+t11*t10;
  double t15 = t10*t10+t12*t12; /* t13 + t14 + t15 = p1.x*p1.x + p1.y*p1.y */
   ...
  t8 = ...
  }
return (t8<0 ? -1 : t8==0 ? 0 : 1 )
}
```

Figure 12: Output generated from the 2d Incircle example; `P2[i]` contains 2^i and `P2x3[i]` contains $3 \cdot 2^i$.