Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry

Steven Fortune
AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`sjf@research.att.com`

Christopher J. Van Wyk
Dept. of Mathematics and Computer Science
Drew University
Madison, New Jersey 07940
`cvanwyk@drew.edu`

March 11, 1996

Abstract

Geometric algorithms are usually described assuming that arithmetic operations are performed exactly on real numbers. A program implemented using a naive substitution of floating-point arithmetic for real arithmetic can fail, since geometric primitives depend upon sign-evaluation and may not be reliable if evaluated approximately. Geometric primitives are reliable if evaluated exactly with integer arithmetic, but this degrades performance since software extended-precision arithmetic is required.

We describe static-analysis techniques that reduce the performance cost of exact integer arithmetic used to implement geometric algorithms. We have used the techniques for a number of examples, including line-segment intersection in two dimensions, Delaunay triangulations, and a three-dimensional boundary-based polyhedral modeller. In general, the techniques are appropriate for algorithms that use primitives of relatively low algebraic total degree, *e.g.*, those involving flat objects (points, lines, planes) in two or three dimensions. The techniques have been packaged in a preprocessor for reasonably convenient use.

1 Introduction

It is convenient to describe geometric algorithms in the "real number RAM" model of computation, with unit-cost operations—including arithmetic, assignment, and comparison—on real numbers. Computer programs that implement these algorithms are often written by naively substituting floating-point operations for real. This replacement is undeniably convenient, and yields a program with very fast numeric operations.

Since floating-point arithmetic is intrinsically approximate, however, the machine's native precision may not suffice for all geometric tests. For example, when a point lies very close to a line, substituting the point coordinates into the line equation can yield a floating-point number whose rounding error exceeds its magnitude. This makes it impossible to tell reliably whether the point lies above, on, or below the line.

There are several techniques to cope with the limited precision of floating-point arithmetic. One is to choose an ϵ, and to treat any quantities that are smaller in absolute value than ϵ as if they were zero. Many examples are known where using such *ad hoc* tolerances causes geometric programs to fail catastrophically[12, Sect. 3.4],[20, Chap. 4]. Another approach is to perform a careful analysis that describes when an algorithm is stable in the face of floating-point errors[16, 24, 30]. The required analysis is quite difficult and has been completed for only a few algorithms.

Many geometric algorithms can be written in terms of integer or rational arithmetic, so careful programmers might consider avoiding floating-point altogether. For all but the simplest geometric computations, however, the required bit-length of the integers will exceed the native machine precision, so this strategy requires using software multiprecision integer arithmetic. The large performance cost of this approach seems to have made it unpopular. For example, Karasick *et al.* report that substituting off-the-shelf software rational arithmetic for every use of floating-point arithmetic in a program to compute planar Delaunay triangulations slowed the program by a factor of about 10^4[23]; Jaillon reports a similar slowdown for a polyhedral modelling program[22].

Thus, floating-point arithmetic is fast but unreliable, while multiprecision integer arithmetic is exact but slow. Several researchers have tried to marry the two kinds of arithmetic to obtain the best features of both. The common theme of these "adaptive" techniques is to evaluate each expression to just enough precision to obtain a reliable answer. Karasick *et al.* report dramatic improvements in performance from combining adaptive-precision arithmetic with a variety of other optimizations specific to the algorithm; eventually they obtained an overall runtime slowdown of about 4 from the floating-point to the adaptive-precision implementation. Using only adaptive-precision techniques, Jaillon obtained a ratio of about 8. Section 3.2 describes some approaches to adaptive precision and measures the runtime overhead contributed by each.

The runtime ratios of adaptive precision arithmetic represent a big improvement over naive exact arithmetic. Still, few programmers will welcome such a large increase in runtime, especially when the slowdown occurs even on problems that rarely need full-precision computation. Implementations of adaptive-precision arithmetic typically exhibit such high overhead because they replace each arithmetic operation, which in floating-point is a single machine instruction, by a subroutine call or a macro expansion that does relatively complicated bookkeeping, often including some form of memory management. Since arithmetic computation typically appears in the inner loop of a geometric computation, this substitution imposes substantial overhead.

We present a strategy that can be applied before compilation to generate efficient adaptive code for multiprecision integer arithmetic. Our strategy statically analyzes entire arithmetic expressions; this makes it especially suitable in the context of computational geometry, where the expressions that define geometric primitives are known in advance. The generated code uses two levels of adaptive precision to compute the sign of an arithmetic expression. First, it evaluates the expression in floating-point arithmetic. If the magnitude of the floating-point result exceeds an error bound, then the sign of the floating-point value is the true sign of the expression. The error bound is computed by static analysis of the structure of the expression and estimates of the sizes of the operands; thus, testing the magnitude of the floating-point approximation against the error bound requires two comparisons at runtime. If the floating-point approximation is smaller in magnitude than the error bound, the expression is evaluated exactly using code generated from the static analysis. Section 4 includes more details of the static analysis.

We have packaged this strategy in a preprocessor called LN. (LN stands for "little numbers," to emphasize that the preprocessor is not a general-purpose "big number" package.) The input to LN is a set of geometric primitives, each expressed in integer polynomials. The output of LN is C++ code that simulates exact evaluation of the primitives. Section 5 describes LN.

Section 6 presents some results from experiments with programs that use LN to implement several geometric algorithms, including Delaunay triangulation, planar line segment intersection, and three-dimensional polyhedral modeling. On generic problem instances, the primitives generated by LN impose very little runtime penalty: the exact-arithmetic version runs almost as fast as a floating-point version of the program. On degenerate instances contrived to require many exact evaluations, the runtime penalty typically has been less than a factor of two. The exact penalty depends heavily upon the details of the algorithm and the problem instance.

Moving from the real number RAM version of an algorithm to an integer version suitable for programming is harder than simply changing the definitions of "number" types. First, it must be possible to represent geometric data using only integers. Next, the programmer must define primitives carefully, paying particular attention to the bit-length required to evaluate them exactly. The primitives for some algorithms are well understood, while other algorithms may require considerable ingenuity, particularly to reduce the necessary bit-length. (The details of an example using circles appear in [33].) Third, the programmer should consider which parts of a computation do not require exact arithmetic at all, partitioning the program into floating-point and exact-arithmetic pieces. Because of engineering choices made during the design and implementation of LN, using it to implement a geometric algorithm also imposes its own peculiar challenges. Section 7 contains some observations about these issues.

1.1 Other work

Software extended-precision arithmetic has been developed for many applications; for example, Brent's MP package[2] provides arbitrary-precision floating-point arithmetic, while the INRIA BigNum package[29] provides arbitrary-precision integer arithmetic. More recently, attention has focused on the specific needs of computational geometry (Yap and Dubé[35] offer a recent survey).

Several authors have reported experiments with some form of integer (or rational) arithmetic. We have already mentioned the work of Karasick *et al.* using adaptive-precision rational arithmetic. The LEDA algorithms library[27] provides several C++ arithmetic packages, including arbitrary-precision `integer` (see section 3.1) and a variation of adaptive precision called `floatf` (see section 3.2); Mehlhorn and Näher used these LEDA packages to implement the Bentley-Ottmann plane-sweep algorithm[25]. Benouamer *et al.* detail their experience using lazy rational arithmetic to implement the Bentley-Ottmann algorithm and a polyhedral modeller[4, 5, 22].

Sometimes integer or rational arithmetic is not enough to describe geometric primitives conveniently. Yap advocates general computation on algebraic numbers, and has offered a preliminary software design[34]. The LEDA library also includes a `real` package that provides exact $+$, $-$, $\times$, $\div$ and $\sqrt{}$; Burnikel *et al.*[3] measure empirically the precision required for the incircle test on line segments, a predicate best expressed using nested square roots.

Many geometric predicates use the sign of a determinant. Clarkson[9] and Avnaim *et al.*[1] present algorithms to compute the sign of an integer determinant using less arithmetic precision than would be required to represent the value of the determinant. It is not clear how these algorithms compare to direct evaluation in practice.

Since most scientific computation is performed using floating-point arithmetic, the input to a geometric algorithm is often expressed with floating-point values. It would be convenient to perform exact computation on floating-point values directly. Since floating-point numbers are rational, they could be manipulated with software rational arithmetic. Priest[28] gives an alternate technique that uses a purely floating-point representation; he analyzes a few geometric predicates. The representation used in section 4.2 is similar in spirit to Priest's representation.

2 Preliminaries

2.1 Geometric primitives in integer arithmetic

Our methods apply when geometric data can be represented using integer values and geometric primitives can be written as multivariate integer polynomials in those values. Homogeneous coordinates are a convenient way to express rational affine coordinates as integer tuples. The three-dimensional point with homogeneous coordinates (x, y, z, w) has affine coordinates $(x/w, y/w, z/w)$.

Geometric primitives can be partitioned into *predicates*, which determine control flow and require only the sign of a polynomial, and *constructors*, which generate new geometric objects and require the value of a polynomial. A rough estimate of the arithmetic bit-length required to evaluate a polynomial is the

$$\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix} \qquad \begin{vmatrix} x_1 & y_1 & z_1 & w_1 \\ x_2 & y_2 & z_2 & w_2 \\ x_3 & y_3 & z_3 & w_3 \\ x_4 & y_4 & z_4 & w_4 \end{vmatrix} \qquad \begin{vmatrix} x_1 & y_1 & x_1^2+y_1^2 & 1 \\ x_2 & y_2 & x_2^2+y_2^2 & 1 \\ x_3 & y_3 & x_3^2+y_3^2 & 1 \\ x_4 & y_4 & x_4^2+y_4^2 & 1 \end{vmatrix}$$

Figure 1: Geometric predicates: 3d affine orientation (left), 3d homogeneous orientation(center), 2d incircle(right).

product of the bit-length of the coordinate data and the total degree of the polynomial. (Whenever we use the word "degree" below, we mean "total degree.")

Figure 1 includes three examples of predicates. The sign of the orientation determinant tells whether the fourth point lies above, on, or below the oriented hyperplane through the other three points. In d dimensions, the orientation test is a polynomial of degree d for affine coordinates and $d+1$ for homogeneous coordinates. The two-dimensional incircle test tells whether the fourth point lies outside, on, or inside the circle defined by the other three points; for points with affine coordinates in d dimensions, its degree is $d+2$.

As an example of a constructor, we consider the plane through three given points. The plane coefficients are given by the 3×3 subdeterminants of the first three rows of the orientation determinant. For homogeneous point coordinates these coefficients are of degree 3, while for affine coordinates they are a mixture of degrees 2 and 3. By duality, the same constructor gives the point of intersection of three planes. Degree estimates multiply as constructors are composed; for example, the point of intersection of three planes, each defined by three points, has degree 9 in the coordinates of the original points (assuming homogeneous coordinates).

2.2 IEEE floating-point arithmetic

We assume throughout that IEEE double-precision floating-point arithmetic[21] is being used. This means that a floating-point value effectively has a 53-bit mantissa, which implies that the relative error in any floating-point operation is at most 2^{-53}. It also means that floating-point operations are exact so long as the result can be represented exactly.

3 Software Operators

Implementing arithmetic in software means, in effect, replacing every arithmetic operation in an expression like

```
det = sign(a*d-b*c);
```

by invocations of the appropriate subroutine or macro to yield code like

```
t1 = mul(a,d); t2 = mul(b,c); t3 = sub(t1,t2); det = sign(t3);
```

C++ makes this approach particularly convenient, since the arithmetic operators can be overloaded to denote subroutine calls. In this section we consider various alternatives for what happens inside the subroutines, emphasizing the performance implications of each choice.

Suppose that a floating-point geometric program spends a fraction a of its runtime performing geometric computations. If substituting subroutine calls for native floating-point operations increases the time required for those computations by a factor of f, then the overall runtime of the program will increase by a factor of

$$1 - a + af = 1 + (f - 1)a.$$

The fraction a varies widely; when computing Delaunay triangulations in two and three dimensions, we have observed $0.2 \leq a \leq 0.5$[13].

We wrote some short test programs to estimate the factor f for various alternatives. Details of the programs appear in Appendix A. We timed each program as it computed one million 3d homogeneous orientation tests. Since arithmetic accounts for almost all the runtime of the floating-point version of this program, the ratio of the runtime for an alternative version to the floating-point runtime gives a good idea of the magnitude of f for that alternative.

3.1 Universal Exact Arithmetic

Arithmetic operators can be overloaded to replace floating-point operations by exact integer arithmetic using the LEDA `integer` class. The increase in runtime depends on the bit-length of the input coordinates. When the determinant entries are initialized with random 31-bit integers, runtime increases by about a factor of 60 over floating-point; with random 53-bit integers, it increases by about a factor of 100 over floating-point. We have observed similar performance ratios with other off-the-shelf software libraries for exact integer arithmetic[13].

3.2 Adaptive Precision

The idea of adaptive-precision arithmetic is to evaluate the expressions in a geometric primitive to just enough precision to permit reliable computation. This simple idea admits a rich spectrum of engineering choices. To focus the discussion on overhead, we shall consider only a two-level adaptive-precision strategy for evaluating predicates: first, evaluate the expression in floating-point; if its sign cannot be determined, evaluate the expression using exact arithmetic. We refer to this strategy as "filtered floating-point": we use floating-point to "filter out" easy instances of primitives.

To implement this strategy, each arithmetic operation ($+$, $-$, $\times$) must compute both a result and an error bound on that result, and also must record the operation and its operands in an expression graph for possible later use. We can estimate the relative cost f of this two-level strategy as

$$f = f_E + f_R + xe,$$

where f_E is the cost of accumulating error bounds on floating-point operations and comparing against them, f_R is the cost of recording the operations, x is the fraction of tests that must be evaluated exactly, and e is the overhead of exact evaluation. Notice that the overhead $f_E + f_R$ applies even when exact evaluation is never necessary (*i.e.*, $x = 0$); thus, $f_E + f_R$ is a lower bound on f.

We estimated f_E and f_R separately. Our measurements suggest that $12 \leq f_E \leq 14$ and $24 \leq f_R \leq 33$. We also measured $11 \leq f_E \leq 15$ for LEDA's `floatf` class. Combining these estimates, we conclude that two-level adaptive precision arithmetic is between 35 and 48 times as expensive as floating-point arithmetic when no exact evaluation is required.

3.3 Hand-coded Lazy Evaluation

Since it is so expensive to record an arithmetic expression graph at runtime, a clever programmer might code lazy evaluation by hand. Figure 2 depicts lazy evaluation of 2×2 determinants using classes provided by LEDA. This simple approach is still fairly expensive. Assuming $f_E \approx 11$ (as for LEDA `floatf`) and $a \approx 30\%$

```
int det2x2(integer a, integer b, integer c, integer d)
{
        floatf fa = a, fb = b, fc = c, fd = d;
        int approxresult = Sign(fa*fd - fb*fc);
        if (approxresult != NO_IDEA) return result;
        return sign(a*d - b*c);
}
```

Figure 2: Hand-coded lazy evaluation using LEDA classes `floatf` and `integer`

of total runtime, this substitution will increase the runtime of the program by at least a factor of four, even when exact arithmetic is never needed.

3.4 Other work

The work reported by Karasick *et al.* most closely resembles the hand-coded lazy evaluation of Section 3.3, while the work of Benouamer *et al.* most closely resembles the adaptive-precision techniques of Section 3.2 modified for rational, rather than integer, values. Both sets of authors report performance ratios that seem generally consistent with our observations. More detailed comparison is difficult. Neither group gives enough timing statistics to determine how much runtime was devoted to arithmetic, hence how much overhead their technique imposes, nor does either group give details of the behavior of their implementation on degenerate instances.

4 Static analysis techniques

The set of primitives used by a geometric algorithm is fixed when the program is written. This suggests that it should be possible to reduce the performance costs of the components of adaptive-precision arithmetic with the help of static analysis of the arithmetic expressions. That is, we can use this analysis to move many of the computations associated with adaptive-precision arithmetic from runtime to compilation time.

We show how to analyze an arithmetic expression to obtain static error bounds and to generate efficient code for exact evaluation. The methods apply to any multivariate integer polynomial, *i.e.*, an expression over the operations $\{+, -, \times\}$ with integer constants and variables. To use the methods, we need an upper bound on the bit-length of each variable that appears in the expression. Given such bounds, we can compute bounds inductively on the bit-length of each subexpression:

$$\begin{aligned}\mathbf{maxbitlen}(a \pm b) &= 1 + \max(\mathbf{maxbitlen}(a), \mathbf{maxbitlen}(b)),\\ \mathbf{maxbitlen}(a \times b) &= \mathbf{maxbitlen}(a) + \mathbf{maxbitlen}(b).\end{aligned}$$

4.1 Static error bounds

We can compute a static upper bound **maxerr** on the absolute error in double-precision evaluation of an integer polynomial as follows. If $\mathbf{maxbitlen}(x) \le 53$, then $\mathbf{maxerr}(x) = 0$. Otherwise,

$$\begin{aligned}\mathbf{maxerr}(a \pm b) &= \mathbf{maxerr}(a) + \mathbf{maxerr}(b) + 2^{\mathbf{maxbitlen}(a+b)-53},\\ \mathbf{maxerr}(a \times b) &= \mathbf{maxerr}(a)\,\mathbf{maxbitlen}(b) + \mathbf{maxerr}(b)\,\mathbf{maxbitlen}(a) + 2^{\mathbf{maxbitlen}(a\times b)-53}.\end{aligned}$$

Since **maxerr** depends only on the structure of the expression and the bit-lengths of the variables, it can be computed statically. The runtime error-bound test compares the magnitude of the floating-point value with **maxerr**; this can be implemented in two comparisons.

Since **maxerr** does not take into account the actual values of the variables, it is more conservative than the error bounds in Section 3.2. To estimate how much more conservative, consider the three-dimensional homogeneous orientation determinant (evaluated using dynamic programming), and assume that all matrix entries have 31-bit coordinates. For this situation, the value of **maxerr** is about $2^{80} \approx 1.2 \times 10^{24}$. For sixteen 31-bit integers, chosen randomly from the range $2^{30} \ldots 2^{31}$, the LEDA `floatf` bound is about 1.3×10^{25}, and the dynamic bound (from the standard formulæ) is about 4×10^{21}. (It may help to look at these bounds by rescaling coordinate data from $0 \ldots 2^{31}$ to $[0, 1]$. Dividing by $2^{31\times4} \approx 2.1 \times 10^{37}$, we find a scaled static error bound of about 5×10^{-14}, a scaled `floatf` error bound of about 6×10^{-13}, and a scaled dynamic bound of about 8×10^{-16}.)

The `floatf` bound and the dynamic bound, of course, depend upon the actual magnitudes of the coordinates. If each coordinate in the orientation determinant were divided by 2, these bounds would be reduced by $2^4 = 16$. The static bound, on the other hand, would not change. Thus, the static bound is most effective when the input coordinates do not lie far from their maximum possible value.

4.2 Compiled exact evaluation code

Part of the high performance cost of conventional multiprecision integer arithmetic can be attributed to bookkeeping overhead, including subroutine linkage, memory management, and loops to handle integers of arbitrary bit-length. When every arithmetic operator incurs such costs, the total overhead for operations on the relatively short integers needed by geometric predicates can be quite significant. Most of this overhead can be eliminated by generating evaluation code that is carefully tailored to each expression.

This section describes one scheme that such tailored evaluation code can use. It stores a multiprecision integer as a tuple of double-precision floating-point values. Though it may seem odd to use floating-point arithmetic to implement exact arithmetic on integers, we do so because double-precision floating-point has been extensively optimized on current machine architectures and allows exact operations on 53-bit integers.

A multiprecision integer value a is represented as the unevaluated sum of double-precision floating-point variables

$$a_0 + \cdots + a_m, \text{ where } a_i = a'_i \times 2^{ri},\ a'_i \text{ integral,} \tag{1}$$

and r is the digit (or radix) length. The number of variables depends upon the radix length and the estimated bit-length of the value; any integer up to 2^{1023} can be represented in this way (the limit comes from the limit on exponents in IEEE double precision). A multiprecision value is *normalized* if each a'_i is at most 2^r in absolute value. Integer values need not be normalized, so the representation of a multiprecision value is not unique.

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

$$c_0 := a_0 + b_0, c_1 := a_1 + b_1, \ldots, c_m := a_m + b_m. \tag{2}$$

so long as each result is exactly representable. The multiprecision product $c := ab$ is implemented using the usual $O(m^2)$ algorithm:

$$c_k := \sum_{0 \le j \le m} a_j b_{k-j}, \tag{3}$$

again so long as each result is exactly representable.

primitive	total degree	Operation count fp	31-bit exact	53-bit exact
affine 2d orientation	2	7	35	120
affine 3d orientation	3	23	110	480
affine 4d orientation	4	61	570	1500
plane from three affine 3d points	3	38	468	1135
constructed plane:point dot product	4	6	61	180
homogeneous 2d orientation	3	14	155	500
homogeneous 3d orientation	4	45	645	1380
2d incircle	4	35	455	860
3d incircle	5	87	1010	2360
line through two affine points	2	5	60	145
intersection of two constructed lines	3	10	150	315
coordinate comparison of intersection points	5	4	115	280

Figure 3: Floating-point operation counts. The total degree is expressed in terms of input coordinates. Operation counts for exact evaluation are approximate.

Often with (3) and sometimes with (2), some term in the result cannot be represented exactly as a floating-point value, and the operands must be normalized before the operation. To normalize, we execute the following simultaneous assignment in order of increasing i:

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

This guarantees that for each i, a_i' is at most 2^r in absolute value. Profiling revealed that the library modulus routine is slow. If $a_i < 2^{127}$, an alternative is to add a large constant to align the bit positions in the normalized floating-point significands, perform the mod by converting to single-precision floating-point and subtracting, and finish by subtracting the large constant; scaling is required if the value $a_i \geq 2^{127}$. Another, less portable, alternative would be to use bit operations on the significands.

The choice of the radix bit-length 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 bit-length 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 between normalizations, it is useful to reduce the radix length slightly. The value $r = 23$ appears to be an appropriate compromise, which was used for all timing results reported in this paper.

Figure 3 presents counts of double-precision floating-point operations for three methods of evaluating various primitives: naive floating-point evaluation, exact evaluation on 31-bit integer coordinates, and exact evaluation on 53-bit integer coordinates. For the exact evaluation, each normalization counts as four floating-point operations. The operation counts suggest the cost of evaluation, but they do not predict it exactly. The homogeneous 3d orientation test on 31-bit coordinates takes $14 - 21$ times as long to evaluate the determinant exactly as it takes using double-precision floating-point arithmetic.

5 The LN preprocessor

The static analysis techniques described in Section 4 have the potential to improve substantially the performance of two-level adaptive-precision arithmetic. The LN preprocessor[14] packages these techniques for reasonably convenient use.

```
tuple HPoint3d(x,y,z,w);
macro  det4x4(a:HPoint3d, b:HPoint3d, c:HPoint3d, d:HPoint3d) =
{  dxy = a.x*b.y - b.x*a.y;  dxz = a.x*b.z - b.x*a.z;  dxw = a.x*b.w - b.x*a.w;
   dyz = a.y*b.z - b.y*a.z;  dyw = a.y*b.w - b.y*a.w;  dzw = a.z*b.w - b.z*a.w;
   dxyz = c.x*dyz - c.y*dxz + c.z*dxy;  dxyw = c.x*dyw - c.y*dxw + c.w*dxy;
   dxzw = c.x*dzw - c.z*dxw + c.w*dxz;  dyzw = c.y*dzw - c.z*dyw + c.w*dyz;
   det = d.x*dyzw - d.y*dxzw + d.z*dxyw + d.w*dxyz;
   sign(det)
};

class integer = int<31>, HPoint3d;
proc det4x4;
```

Figure 4: Homogeneous 3d orientation in LN.

The LN preprocessor generates C++ classes and subroutines. A class generated by LN stores integers, while a subroutine generated by LN evaluates an integer polynomial. Thus, the interface to LN, while relatively low-level, is at a higher level of abstraction than a "number" data type and a set of operations. A class that is useful for representing geometric data will probably require both numeric and nonnumeric data; such a class could include an LN-generated class as a data member, or could be derived from an LN-generated class. Similarly, a high-level geometric primitive might need to invoke several subroutines generated by LN.

The input language to LN describes integer polynomial expressions. Primitives include 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 `tuple`s of fixed arity and parametrized expressions called `macro`s. To turn the expressions into C++, the maximum bit-length of any input variable must be specified. LN uses this information to compute bounds on the bit-lengths of all intermediate and output values, to generate error bounds, and to produce exact evaluation code.

Figure 4 contains an LN definition of the 3d homogeneous orientation predicate. The `class` line directs LN to define a C++ class `integer` that can hold 31-bit integers, and another C++ class `HPoint3d` that holds four `integer` coordinates. The `proc` line directs LN to define a subroutine `det4x4()` that returns the sign of the implied 4×4 determinant. Figure 5 contains an edited portion of the C++ code generated by LN.

The code in Figure 5 evaluates the determinant in floating-point; if and only if its magnitude is less than the static error bound, the code evaluates the determinant exactly. This lazy evaluation strategy extends to arbitrary Boolean tests, so a Boolean test in the expression language usually generates two tests in the C++ program: one determines whether the floating-point computation is reliable; the second actually performs the test. The precompiler also avoids generating code that evaluates the same 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.

LN can also be used to define a constructor, which produces new geometric data. Figure 6 gives the LN definition of a primitive that computes the plane through three homogeneous points in R^3. From this definition, LN generates a declaration for class `Plane`, one of whose member functions is a constructor generated from the LN macro `plane`. If `a`, `b`, and `c` are variables of class `HPoint3d`, the programmer could define and initialize an instance of `Plane` by writing

```
Plane pi(a,b,c);
```

```
int det4x4(const HPoint3d& a, const HPoint3d& b, const HPoint3d& c, const HPoint3d& d){
t[1] = a.z*b.w-b.z*a.w;
t[2] = a.y*b.w-b.y*a.w;
t[3] = a.y*b.z-b.y*a.z;
t[4] = a.x*b.w-b.x*a.w;
t[5] = a.x*b.z-b.x*a.z;
t[6] = d.x*(c.y*t[1]-c.z*t[2]+c.w*t[3])-d.y*(c.x*t[1]-c.z*t[4]+c.w*t[5]);
t[7] = a.x*b.y-b.x*a.y;
t[8] = t[6]+d.z*(c.x*t[2]-c.y*t[4]+c.w*t[7])+d.w*(c.x*t[3]-c.y*t[5]+c.z*t[7]);
if (-P2[86]<t[8] && t[8]<P2[86]) {
  t[9] = d.x;               // splits d.x into low and parts
  t[10] = float(P2x3[45]+t[9])-P2x3[45];
  t[9] -= LNt[10];
  ...                       // accumulate determinant as sum of f.p. numbers
  t[8] = (((((t[134]+t[133])+t[132])+t[131])+t[130])+t[129]); // sign of sum is correct
}
return t[8]<0 ? -1 : t[8]>0 ? 1 : 0;
}
```

Figure 5: Edited C++ subroutine produced by LN from Figure 4. `P2[i]` contains 2^i and `P2x3[i]` contains 3×2^i. Necessary conversions from `int` to `double` have been omitted for clarity.

The plane coefficients are computed exactly. Since each of them is about 97 bits long, they do not fit into a native C++ type. LN generates an appropriate representation, which is not usefully accessible within the C++ program. An instance of `Plane`, however, may appear as a parameter to another subroutine generated by LN, such as `beyond` in Figures 6 and 7.

LN associates information about maximum bit-lengths with each class it generates. This means that the programmer need not know the actual bit-length of `Plane` coefficients. It also means that another "plane" with different maximum bit-length would have to be an instance of a different C++ class. Section 7.2 discusses the reasons for and some consequences of this tight association between bit-length and class.

6 Experimental Results

We have used LN to implement several geometric algorithms. This section describes the performance of the resulting programs. Section 7 describes our experience coding the algorithms.

One measure of performance would be the relative cost of floating-point and LN implementations of a geometric algorithm. We did not, however, implement each algorithm separately using floating-point and LN. Instead, we can estimate the extra performance cost of the LN version as the cost of the floating-point filter plus the cost of any required exact evaluations of primitives.

The total cost of exact primitive evaluations is determined by the cost of each exact primitive and the number of times each is evaluated. The cost of a primitive depends only on the algorithm and bit-length, and can be determined relatively easily (see Figure 3). The number of exact evaluations required depends on the distribution of data and the effectiveness of the floating-point filter. Random data requires very few exact evaluations; degenerate or nearly degenerate data may require many exact evaluations.

We report the behavior of each algorithm on a wide range of data sets, from highly degenerate to random. Behavior on real-world data should lie somewhere in this range; real-world data might well be

```
tuple HPoint3d(x,y,z,w), Plane(a,b,c,d);
macro  plane(a:HPoint3d, b:HPoint3d, c:HPoint3d) =
{  dxy = a.x*b.y - b.x*a.y;  dxz = a.x*b.z - b.x*a.z;  dxw = a.x*b.w - b.x*a.w;
   dyz = a.y*b.z - b.y*a.z;  dyw = a.y*b.w - b.y*a.w;  dzw = a.z*b.w - b.z*a.w;
   dxyz = c.x*dyz - c.y*dxz + c.z*dxy;  dxyw = c.x*dyw - c.y*dxw + c.w*dxy;
   dxzw = c.x*dzw - c.z*dxw + c.w*dxz;  dyzw = c.y*dzw - c.z*dyw + c.w*dyz;
   Plane(dxyz, -dxyw, dxzw, -dyzw)
};
macro beyond(pi:Plane, p:HPoint3d) = sign(pi.a*p.x + pi.b*p.y + pi.c*p.z + pi.d*p.w);

class integer = int<31>, HPoint3d, Plane(-);
proc Plane::Plane = plane;
proc beyond;
```

Figure 6: LN program to calculate the plane through three points.

```
class HPoint3d {public: int x,y,z,w; };
class extended97 { public: double v0,v1,v2,v3,v4; };
class Plane { public:
        extended97 a,b,c,d;
        Plane(){};
        Plane(const HPoint3d&, const HPoint3d&, const HPoint3d&);
};
int beyond(const Plane&, const HPoint3d&);
```

Figure 7: LN-generated C++ signatures for the example in Figure 6 (slightly edited).

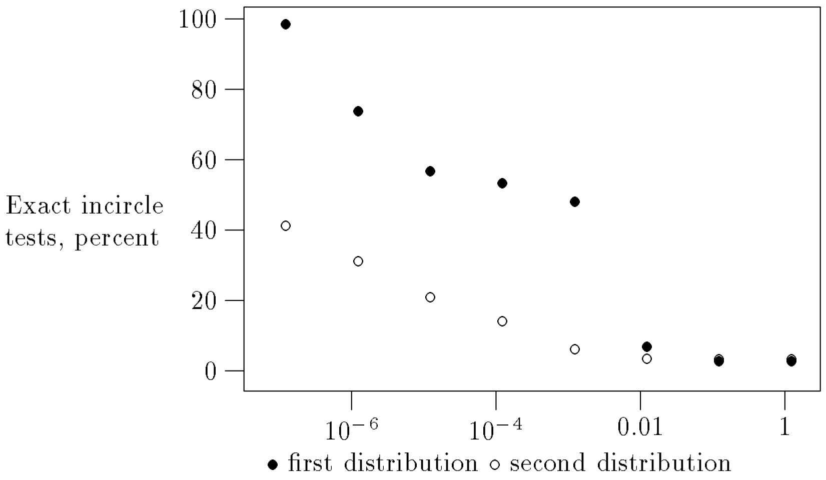

Figure 8: Percentage of incircle tests that required exact arithmetic in divide-and-conquer planar Delaunay triangulation, as a function of interpolation parameter ϕ.

generated by a non-random process, and so contain many degeneracies.

6.1 Delaunay Triangulations

We experimented with the divide-and-conquer algorithm for planar Delaunay triangulations[18]. The required primitives are the orientation test and incircle test (both in two dimensions). If points are represented with affine coordinates, these primitives have degrees 2 and 4, respectively.

Each input data set consisted of 10000 points with 53-bit integer coordinates. Input sites were chosen from two families of distributions; each family depends upon an interpolation parameter $\phi \in [0, 1]$. The ith site in the first family was chosen by

$$p_i = \phi r_i + (1 - \phi) c_i$$

where r_i has 53-bit integer coordinates chosen uniformly at random and c_i is an integer point that lies approximately on the circle of radius 2^{53} around the origin (*i.e.*, each coordinate was computed in double-precision floating-point using trigonometric functions, then rounded to integer). When $\phi = 1$, the sites are random; as ϕ approaches zero, the sites become more nearly cocircular. The second family of distributions is similar, except that c_i was chosen from a circle of radius $2^{53}/16 = 2^{49}$ centered at $(15 \times 2^{49}, 15 \times 2^{49})$.

Figure 8 plots the percentage of incircle tests that required exact arithmetic as a function of the interpolation parameter ϕ. While the percentage of exact tests increases as ϕ decreases, the total running time also decreases with ϕ. This happens because the total number of incircle tests also decreases, from about 20 per site at $\phi = 1$ to about 5 per site at $\phi = 10^{-7}$. The percentage of time devoted to incircle tests ranges from about 25% at $\phi = 1$ to a maximum of about 55% at $\phi = 10^{-4}$.

We report on experiments with the flipping algorithm in two dimensions and the random-incremental algorithm in three dimensions in [13].

6.2 Segment Intersection

The Bentley-Ottmann plane-sweep algorithm[6] reports all intersections in a set of line segments. Conceptually, the algorithm maintains the state of a vertical sweepline as it moves across the plane; the state changes

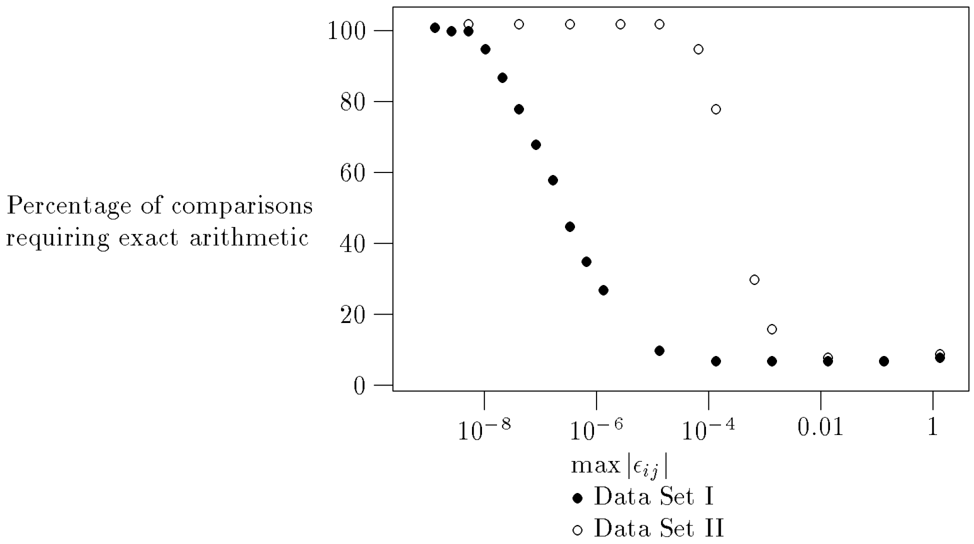

Figure 9: Exact coordinate comparisons required for Bentley-Ottmann plane-sweep, as a function of relative perturbation ϵ.

at events where a segment starts or ends or two segments cross.

Profiling our implementation[32] revealed that it spends over half its time maintaining the event queue. This requires comparing the coordinates of points, of which there are two kinds. Segment endpoints are represented using affine coordinates, while intersection points are naturally represented using homogeneous coordinates. Our program stored all intersection points, which meant their coordinates had to be computed to full precision; this took about 10% of the total runtime. Subsequent coordinate comparisons, however, use the floating-point filter provided by LN.

We experimented with several data sets of 250 segments whose endpoint coordinates have bit-length 31. If the segments were chosen uniformly at random, then only about 5% of the coordinate comparisons required exact arithmetic; altogether, arithmetic operations accounted for about 33% of runtime. Two other sets of example data illustrate the performance cost of degenerate input. Data Set I started with the 250 segments

$$\{(0, 10^9 \times j/250), (10^9, 10^9 \times (250 - j)/250) : j = 0, \ldots, 249\}$$

(i.e., 250 segments crossing at a single point), then perturbed each endpoint coordinate independently by a random integer chosen in the interval $-\epsilon 10^9 \ldots \epsilon 10^9$. Data Set II consisted of 250 copies of the segment $((0, 10^9), (10^9, 10^9))$, with each endpoint perturbed as for Data Set I.

Figure 9 gives the percentage of intersection-point coordinate comparisons that require exact arithmetic as a function of ϵ. The total running time for Data Set I increased by about 30% as ϵ decreased from 1.024 to 10^{-9}, with arithmetic eventually accounting for about 45% of the runtime. Note that once ϵ becomes smaller than 10^{-3}, the number of intersections remains constant.

As ϵ decreased from 1.024 to 10^{-9} on Data Set II, the total running time more than doubled, as did the number of intersections between segments.

In some applications it is desirable to round the intersection points to have the same bit-length as the original data; our implementation does not do this. Such rounding can be accomplished by modifying the Bentley-Ottmann algorithm[17, 19, 26].

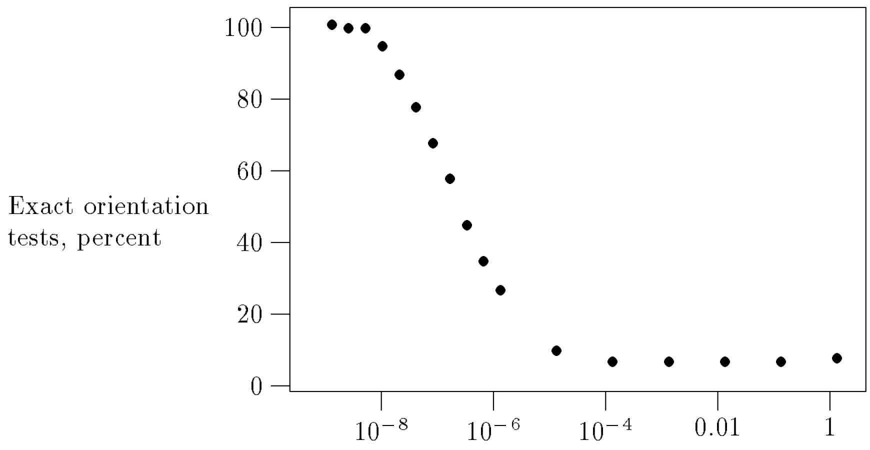

Figure 10: Exact orientation tests required for polyhedral intersection, as a function of rotation angle θ.

6.3 Polyhedral modelling

A polyhedral modeller[20] supports Boolean operations on three-dimensional polyhedral solids, as well as affine transformations such as rotation and translation. We implemented a bare-bones boundary-representation modeller, starting with ideas of Sugihara and Iri[31]. More details of the algorithms and implementation appear elsewhere[15].

Plane equations are the primary geometric information used to represent a polyhedron. A Boolean operation on two polyhedral solids introduces no new face planes, hence no new primary geometric information. A vertex is defined as the point of intersection of three planes.

A transformation on a polyhedral solid is effected by multiplying plane coefficients by a transformation matrix with integer entries. This multiplication increases the bit-length of the plane coefficients; since this increase is undesirable, the coefficients are rounded to a specified maximum bit-length. Geometrically, the rounding perturbs the plane slightly, which may cause inconsistencies between geometric and combinatorial information. An important component of the implementation is an algorithm that reconstructs consistent combinatorial information after the face planes have been rounded[15].

The orientation test on planes suffices to implement the modeller. In effect, this test decides whether the point of intersection of three oriented planes lies above, on, or below a fourth oriented plane; it is dual (and mathematically identical) to the orientation test on homogeneous points in three dimensions. As a performance optimization, the program computes and caches vertex locations. This means that most orientation tests, which involve a vertex and a fourth plane, can be replaced by a cheaper dot-product of vertex and plane coefficients.

For the experiment reported here, we used 31-bit integers for face-plane coefficients. This implies that each (homogeneous) vertex coefficient has absolute value at most about 2^{100}. The exact computation of vertex coefficients contributes about 7% to total runtime.

We chose a convex polyhedron with about 250 sides (obtained by intersecting randomly-oriented unit cubes centered at the origin). We intersected the polyhedron with a copy of itself rotated by an angle θ, for θ varying between 10^{-1} and 10^{-9} radians. Figure 10 plots the percentage of the orientation tests that had to be performed exactly, as a function of θ. The percentage of required exact dot-products also increased as θ decreased, though not as quickly. For large θ, arithmetic occupied about 30% of total runtime; this increased to somewhat more than 50% at $\theta = 10^{-9}$, mostly due to the large number of exact orientation tests. Total runtime increased by about 50% from large θ to small θ.

7 Programming Experience

The preceding sections show that LN can be used to implement geometric algorithms expressed using integer arithmetic as reliable programs that run almost as fast as naive floating-point implementations. Unfortunately it is not entirely straightforward to obtain such good performance; it is certainly harder than simply replacing the "number" types in a hypothetical real-arithmetic implementation. The difficulties can be attributed both to formulating a geometric algorithm in terms of integer arithmetic and to using LN to write the program.

7.1 Geometric algorithms with integer arithmetic

The first task in writing a geometric program is deciding which parts require exact arithmetic. Exact arithmetic guarantees geometrically consistent answers to arbitrary geometric tests on arbitrary input data. Not all geometric algorithms, however, require such complete consistency. Since exact tests are potentially expensive, it is appropriate to avoid exact computation when possible. This is particularly relevant for constructed geometric data, where bit-length increases multiplicatively as constructors are applied.

The Delaunay triangulation and the Bentley-Ottmann algorithm each compute some geometric objects whose bit-length is a small multiple of the bit-length of the input data, so it was appropriate to assume exact arithmetic everywhere in those programs. Implementing the polyhedral modeller offered more choices. For our program, each Boolean operation is guaranteed to produce the mathematically defined exact result. The exact results of affine transformations, however, must be rounded to prevent coordinates from growing arbitrarily large. Each of these transformations returns an approximation to the mathematically defined exact result; there is no guarantee that applying a transformation followed by its inverse will yield the original solid. This appears to be a reasonable tradeoff between performance and complete mathematical consistency.

Once the exact-arithmetic part of a geometric algorithm has been identified, we can choose the representation of data and the required geometric primitives. The performance cost of a geometric primitive is largely determined by its degree and the bit-lengths of input data values. Minimizing this cost can require choosing among alternatives that interact in complicated and perhaps unfamiliar ways. Each of our implementations uses a representation that minimizes the degree of the required primitives.

The Delaunay triangulation algorithms were particularly easy to implement, since the required predicates have been worked out in detail. Points are represented using affine coordinates. Using a homogeneous representation would double the degree of the incircle primitive.

The Bentley-Ottmann algorithm requires exact computation of intersection points and exact coordinate comparison to maintain the event queue, and exact comparison of segments along a vertical line to maintain the sweepline. We used affine coordinates to store the endpoints of the input line segments, and homogeneous coordinates to store intersection points between segments. The highest degree of any predicate is 5, which is attained when comparing two intersection points. If the segment endpoints had also had homogeneous coordinates, the highest degree would have been 8.

The polyhedral modeller uses plane equations to represent polyhedral boundaries. It could have used vertex coordinates instead as the primary geometric information. Boolean operations still would not introduce any new face planes, and the plane orientation primitive would still suffice to implement all necessary geometric tests. Plane coordinates, however, would have bit-length about three times that of vertex coordinates, so total bit-length bounds would triple.

In each of these three examples, the choice of representation and primitives depends strongly upon such factors as where a program spends most of its runtime, what kinds of arithmetic and other operations are especially fast or slow on a particular machine, and the quality of the code produced by a particular compiler. All of these practical considerations affect which of several possible representations will yield the best program when implemented. Moreover, once a representation has been chosen, converting to a different representation, even one that is mathematically equivalent in real arithmetic, may require either extensive changes to the program or expensive computations at runtime.

7.2 Expressing algorithms with LN

The LN interface to C++. Ideally, subroutines generated by LN would correspond exactly to low-level geometric primitives. This produces some tension between enriching the expression language so it can express geometric predicates, and keeping the expression language small so as to simplify the implementation of LN. We chose to concentrate on arithmetic: LN's arithmetic expression language is quite versatile and allows fine control of generated primitives. Support for Boolean case analysis, however, is minimal: complicated case analyses should be written in C++.

Take the test that decides whether two nondegenerate planar segments ab and cd intersect. This can be resolved using at most four 2d orientation tests: test whether a and b lie on opposite sides of line cd and similarly if c and d lie on opposite sides of line ab. The orientation test is easily coded in LN. C++ is more appropriate for the intersection test, however, since it requires a case analysis of results from various orientation tests. If the segments can be degenerate, it is even more important to partition the case analysis between LN and C++, because the computation in this case requires a more intricate case analysis and another numeric test to order three points lying on a common line.

As a more complex example, consider the coordinate comparison test required for the Bentley-Ottmann algorithm: lexicographically ordering points, first by x-coordinate, then by y-coordinate. If the affine coordinates of two points `p` and `q` are stored in a native C++ number type (*i.e.*, they are endpoints of input segments), then the lexical ordering could be coded in C++ as

```
lexorder = sign(p.x-q.x); if (lexorder==0) lexorder = sign(p.y-q.y);
```

(assuming `sign` returns -1, 0, or 1). If the two points are each intersection points of segments, then they are represented in homogeneous coordinates, say x, y, w, and the affine coordinates x/w, y/w are rational. Hence the comparison requires cross-multiplication. Furthermore, the coordinates are not a native C++ number type, so only functions generated by LN can access them.

There are several ways one might express this test using LN. One is to have LN generate two subroutines, `xorder(p,q)` and `yorder(p,q)`, which would be called instead of `sign` in the lexical-ordering code above. Function `xorder` would compare two homogeneous points by their affine x-coordinates ($p.x/p.w$: $q.x/q.w$), whereas `yorder` would compare the points by y-coordinates. Another is to have LN generate a single subroutine `rationalorder(a,b,c,d)` that performs the comparison a/b : c/d, which would be called successively with with $p.x, p.w, q.x, q.w$, then with $p.y, p.w, q.y, q.w$. This approach, however, requires the C++ client to expose the internal details of the homogeneous representation of p and q, and involves considerable argument passing. The solution that best captures the geometric meaning of the test is to package all of `lexorder` as an LN-generated subroutine; the required case analysis is minimal. (This example, in fact, motivated the inclusion of any Boolean operators at all within the LN expression language.)

Lazy constructors. The LN two-level evaluation strategy does not extend to constructed objects. A constructed object is always evaluated exactly, although subsequent predicates on a constructed object are first performed in floating-point, using the floating-point rounding of the exact object. We considered extending the two-level evaluation strategy to constructors as well. For the examples that we tried, however, exact evaluation of constructed objects contributes only a small percentage of total runtime; the real benefit of LN comes from lazy evaluation of subsequent predicates. For more complex algorithms that involve cascaded constructions, lazy constructor evaluation might well prove advantageous.

Class distinction by static bit-length. LN determines a maximum bit-length bound for each data member of each class that it generates. The bound is used to compute error bounds, to generate exact evaluation code, and also to determine a static storage representation for integers up to that bound. Thus, changing a bit-length bound requires defining a new class. In the Bentley-Ottmann example, an input point, whose coordinates have short bit-length, is stored in a different class from the intersection point of two segments, whose coordinates have much longer bit-length. In general, every function that constructs a point may result in a different bit-length bound, and hence require a new class.

The profusion of classes to hold "different-sized" objects of the "same geometric type" does not arise at all when algorithms are written with real numbers, and it certainly complicates programming. We originally had two reasons for this design. First, the tight association between class and bit-length allows LN to generate efficient filtering and exact evaluation code, as described in Section 4. Second, the potential cost of an operation on a geometric object varies enormously with bit-length (see Figure 3); distinguishing classes by bit-length forces the programmer to be aware that operations that look logically identical may in fact have very different costs.

So far we have been able to tolerate the enforced class distinctions. Only the Bentley-Ottmann algorithm requires an operation on pairs of points, where either point can be an input or a constructed intersection point. The implementation uses double-dispatching to determine which LN function should be called. The additional complexity caused by the class distinctions is manageable since there are only two point classes.

The enforced class distinctions become very unattractive for more complex algorithms, with many constructors for the same geometric object[8]. It would be desirable to obtain the performance advantages of LN while allowing bit-length to vary. Both error bounds and exact-evaluation code would have to be determined by the actual magnitude of operands, rather than worst-case estimates. Of course, such dynamic computation gave rise to the unattractive cost estimates in Section 3. We speculate that the costs can be reduced considerably by compile-time processing of an entire arithmetic expression, much as LN already does for static bit-lengths.

8 Conclusion

We began this work hoping to get the reliability of exact arithmetic without having to pay (much) for it. Replacing all floating-point by general-purpose exact arithmetic proved unthinkably expensive, but adaptive-precision arithmetic reduced the overhead to the realm of the possible. Between the adaptive precision of Section 3.2 and LN there lies a spectrum of implementation possibilities, each of which has its own advantages and disadvantages. LN moves as much analysis and computation as possible to precompilation and compilation, which reduces the amount of work at runtime, but also reduces the flexibility available to the programmer. Hand-coded lazy evaluation using LEDA `floatf` and `integer` (Section 3.3) does more work at runtime, but is more flexible.

Our methods apply when an algorithm has been written in terms of integer arithmetic. Except for Section 7.1, we have not said much about the difficulties one may encounter while switching from real to integer arithmetic. Researchers in geometry are accustomed already to devoting close attention to the choice of data representation so that certain operations are more or less convenient[20]. Our work suggests that considerations of bit-length need to be weighed with other factors when choosing how to represent geometric objects.

C++ offers a variety of means intended to make better abstractions available to a programmer. There are, however, well founded concerns about the costs of some of these techniques, especially those that require memory management or copying large temporary variables[7]. Our interface to exact arithmetic is less convenient than one that overloads each operator to do more work. But that very inconvenience can serve the programmer as a not-so-subtle reminder that an operation is expensive, and thus can encourage the programmer to recast the computation to reduce or even eliminate the operation.

Here are the steps we recommend for users beset by precision problems. (1) Try the LEDA `real` class. When you find its performance unbearable, (2) move to integer arithmetic. This may be moderately inconvenient, but you can hand-code lazy evaluation with filtering. If performance is still a problem, (3) move to LN. This may require more programming effort, but the final program will be very efficient.

References

[1] F. Avnaim, J.-D. Boissonat, O. Devillers, F. P. Preparata, M. Yvinec, Evaluation of a method to compute the sign of determinants, *Proc. of the Eleventh Ann. Symp. on Comp. Geom.*, C16–C17, 1995.

[2] R. P. Brent, ALGORITHM 524: MP, a Fortran multiple-precision arithmetic, *ACM Trans. on Math. Software*, 4:71-81, 1978.

[3] C. Burnikel, K. Mehlhorn, S. Schirra, How to compute the Voronoi diagram of line segments: theoretical and experimental results. *Proc. 2nd Eur. Symp. Alg. (ESA 94)*, 1994.

[4] M. O. Benouamer, P. Jaillon, D. Michelucci, J.-M. Moreau, A "lazy" solution to imprecision in computational geometry, *Proc. Fifth Canad. Conf. Comp. Geom.*, 73–78, 1993.

[5] M. O. Benouamer, D. Michelucci, B. Peroche, Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation, *Computer-Aided Design* 26(6):403–416.

[6] J. L. Bentley, T. A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.* **28**:643–647, 1979.

[7] T. Cargill, C++ Programming Style, Addison-Wesley, 1992.

[8] J. Chang, V. Milenkovic, An experiment using LN for exact geometric computations. *Proc. 5th Canad. Conf. Comp. Geom.*, 67–72, 1993.

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

[10] T. Dubé, C. K. Yap, A basis for implementing exact geometric algorithms, manuscript, 1994.

[11] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[12] S. Fortune, Progress in computational geometry, Chap. 3 in *Directions in Geometric Computing*, R. Martin, ed., Information Geometers Ltd, 1993, 81–128.

[13] S. Fortune, C. Van Wyk, Efficient exact arithmetic for computational geometry, *Proc. Ninth Ann. Symp. Comp. Geom*, pp. 163–172, 1993.

[14] S. Fortune, C. Van Wyk, LN users manual, manuscript, AT&T Bell Laboratories, 1993.

[15] S. Fortune, Polyhedral modelling with exact arithmetic, *Proc. Third Symp. Solid Modeling and Applications*, pp. 225-234, 1995.

[16] S. Fortune, Numerical stability of algorithms for 2d Delaunay triangulations, *International Journal of Computational Geometry and Applications*, **5**(1,2), 193–213.

[17] D. Greene, F. Yao, Finite-resolution computational geometry, *Proc. 27th Ann. Symp. Found. Comp. Science*, 143–152, 1986.

[18] 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.

[19] J. Hobby, Practical segment intersection with finite precision output, 1993, submitted.

[20] C. Hoffmann, *Geometric and Solid Modelling: an Introduction*, Morgan Kauffmann, 1989.

[21] IEEE standard for radix-independent floating point arithmetic, ANSI/IEEE std 854-1987, IEEE, New York, NY, 1987

[22] P. Jaillon, Proposition d'une arithmétique rationnelle paresseuse et d'un outil d'aide à la saisie d'objets en synthèse d'images, Thèse, Ecole Nationale Superieure des Mines de Saint-Etienne, 1993. URL: `http://www.emse.fr/pub/papers/LAZY/thesePJ.ps.Z`.

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

[24] Z.Li, V. Milenkovic, Constructing strongly convex hulls using exact or rounded arithmetic, *Proc. Sixth Ann. Symp. Comp. Geom*, 235–243, 1990.

[25] K. Mehlhorn, S. Näher, Implementation of a sweepline algorithm for the straight line segment intersection problem, manuscript, 1994.

[26] V. Milenkovic, Rounding face lattices in the plane, *First Canad. Conf. Comput. Geom.*, 1989.

[27] S. Näher, The LEDA user manual, Version 3.1, January 16, 1995. LEDA is available by anonymous FTP from `ftp.mpi-sb.mpg.de` in directory `/pub/LEDA`.

[28] D. M. Priest, On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations, Ph.D. dissertation, U. C. Berkeley, 1993.

[29] B. Serpette, J. Vuillemin, J. C. Hervé, BigNum: a portable and efficient package for arbitrary-precision arithmetic. Research Report 2, Digital Paris Research Laboratory, May, 1989.

[30] K. Sugihara, M. Iri, Construction of the Voronoi diagram for one million generators in single precision arithmetic, First Canadian Conf. on Comput. Geom., Montreal, Canada, 1989.

[31] K. Sugihara, M. Iri, A solid modeling system free from topological inconsistency, *J. Inf. Proc., Inf. Proc. Soc. of Japan* **12**(4): 380–393, 1989.

[32] C. Van Wyk, Plane sweep with efficient exact arithmetic, manuscript, 1994.

[33] C. Van Wyk, Missing real numbers, *Amer. Math. Monthly* **105**:260–265, 1995.

[34] C. Yap, Towards exact geometric computation, *Proc. Fifth Can. Conf. Comp. Geom.*, 405–419, 1993.

[35] C. Yap, T. Dubé, The exact computation paradigm, 452–492, *Computing in Euclidean geometry*, D.Z. Du, F. Hwang, eds, World Scientific, 1995, second edition.

A Timing Alternative Strategies

We used the 3d homogeneous orientation test as a typical floating-point geometric primitive. Figure 1 shows how it can be expressed as the sign of a 4×4 determinant. Function `det4x4()` in Figure 11 uses dynamic programming to evaluate this determinant: evaluate all 2×2 subdeterminants of the first two rows, then all 3×3 subdeterminants of the first three rows, and finally compute the actual determinant. The whole computation requires 28 multiplications and 17 additions or subtractions.

```
struct HPoint3d {
        Number x,y,z,w;
        HPoint3d(double x, double y, double z, double w)
            : x(x), y(y), z(z), w(w) {};
};

int det4x4(const HPoint3d& a, const HPoint3d& b,
           const HPoint3d& c, const HPoint3d& d)
{
        Number dxy = a.x*b.y - b.x*a.y;
        Number dxz = a.x*b.z - b.x*a.z;
        Number dxw = a.x*b.w - b.x*a.w;
        Number dyz = a.y*b.z - b.y*a.z;
        Number dyw = a.y*b.w - b.y*a.w;
        Number dzw = a.z*b.w - b.z*a.w;
        Number dxyz = c.x*dyz - c.y*dxz + c.z*dxy;
        Number dxyw = c.x*dyw - c.y*dxw + c.w*dxy;
        Number dxzw = c.x*dzw - c.z*dxw + c.w*dxz;
        Number dyzw = c.y*dzw - c.z*dyw + c.w*dyz;
        Number det = d.x*dyzw - d.y*dxzw + d.z*dxyw - d.w*dxyz;
        return sign(det);
}

void main()
{
        HPoint3d a, b, c, d; // initialized pseudo-randomly
        for (int i = 0; i < 1000000; i++)
                int ans = det4x4(a,b,c,d);
}
```

Figure 11: Key definitions for timing loop

The program in Figure 11 computes one million 3d orientation tests. To reduce the amount of work the program does besides arithmetic, `det4x4()` takes four (rather than sixteen) arguments by constant reference, and the test is repeated on the same four points.

First we timed the program in Figure 11 when `Number` was defined to be `double`, *i.e.*, using double-precision floating-point arithmetic. To estimate f for universal exact arithmetic, we defined type `Number` to be a LEDA `integer`.

To estimate f_E, we defined type `Number` and overloaded the arithmetic operators as shown in Figure 12. The standard formulæ for the error in floating-point operations are

$$\begin{aligned} \mathrm{error}(a \oplus b) &= \mathrm{error}(a) + \mathrm{error}(b) + \epsilon|a \oplus b|, \\ \mathrm{error}(a \otimes b) &= \mathrm{error}(a)|b| + \mathrm{error}(b)|a| + \epsilon|a \otimes b|, \end{aligned}$$

where $\epsilon = 2^{-53}$ for IEEE double-precision floating-point. Thus, the sign of a computed floating-point value is correct if its magnitude exceeds the error estimate. The code in Figure 12 incorporates two optimizations to the standard formulæ. It defers the multiplication by ϵ until the error bound is used to compute the sign. It also saves the absolute value of each floating-point value, which eliminates two function calls when computing the error bound for a product.

```
struct Number {
        double value, absvalue, error;
        Number(double value) : value(value), absvalue(fabs(value)),
                error(0.) {}
        Number(double value, double absvalue, double error)
                : value(value), absvalue(absvalue), error(error) {}
};

inline Number operator+(const Number& l, const Number& r)
{
        double sum = l.value + r.value, sumabs = fabs(sum);
        return Number(sum, sumabs, l.error + r.error + sumabs);
}

inline Number operator*(const Number& l, const Number& r)
{
        double prod = l.value*r.value, prodabs = fabs(prod);
        return Number(prod, prodabs,
            l.absvalue*r.error + r.absvalue*l.error + prodabs);
}

inline int sign(const Number& op)
{ return (op.absvalue > epsilon*op.error) ? (op.value < 0. ? -1 : 1) : fail(); }
```

Figure 12: Excerpts from definition of `Number` to accumulate error bounds

Figure 13 shows part of the code we used to estimate f_R. Each arithmetic operation allocates a `Record` that holds the arithmetic opcode and pointers to the `Record`s for its left and right operands. Since the lifetime of a `Record` is not predictable, the code uses reference counts to decide when to reclaim a `Record`. Operators `new` and `delete` are also overloaded to make `Record` allocation and deletion be simple list operations.

Timings for the execution of the program in figure 11 on a variety of machines appear in figure 14. Each row indicates a different substitution for the class `Number`.

```
enum Opcode {Add, Subtract, Multiply, Constant};

struct Record {
        Record* lop, *rop;
        Opcode opcode;
        int refcnt;
        void ref() { refcnt ++; }
        void deref() {
                if((refcnt-=1) == 0) {
                        if (opcode != Constant)
                                { lop->deref(); rop->deref(); }
                        delete this; } }
        Record(Record* lop, Record* rop, Opcode opcode)
                : lop(lop), rop(rop), opcode(opcode), refcnt(1) {}
        void* operator new(size_t) {
                Record* ans = freelist;
                return (ans != 0) ? (freelist=ans->lop, ans) : alloc(); }
        void operator delete(void* r) {
                ((Record*)r)->lop = freelist; freelist = ((Record*)r); }
};

struct Number
{
        Record* record;
        Number() { record = new Record(0,0,Constant); }
        Number(double) { record = new Record(0, 0, Constant); }
        Number(const Number& l, const Number& r, Opcode opcode) {
                record = new Record(l.record, r.record, opcode);
                l.record->ref(); r.record->ref(); }
        Number(const Number& arg) { record = arg.record; record->ref(); }
        Number& operator=(const Number& rhs) {
                rhs.record->ref();
                record->deref();
                record = rhs.record;
                return *this; }
        ~Number() { record->deref(); }
};

inline Number operator+(const Number& l, const Number& r)
{ return Number(l, r, Add); }
```

Figure 13: Excerpts from definitions to save state

Arithmetic type	SGI 40 Mhz	SGI 150 Mhz	SGI 200 Mhz	DEC Alpha 3000/400	DEC station 5000
double-precision floating point	7.5	2.0	1.6	2.1	14.6
Error bounds (figure 12)	93	25.7	19	29	169
LEDA `floatf`	87	22.2	16.7	32.3	156
State saving (figure 13)	183	67	53		464
LN exact (31 bit operands)	111	43.6	33	42.2	200

Figure 14: Times in seconds for a million evaluations of the 3d homogeneous orientation test. SGI clock rates of 150 Mhz and 200 Mhz are effective rates (two instructions are issued per major cycle, with major cycle rates of 75Mhz and 100Mhz, respectively).