*Proceedings of the Fifth Canadian Conference on Computational Geometry*, A. Lubiw and J. Urrutia Eds.,
University of Waterloo, Ontario, Canada, August 5–9, 1993, pp. 67–72.

# An Experiment Using LN for Exact Geometric Computations

Jacqueline D. Chang    Victor Milenkovic*
Harvard University Division of Applied Sciences
Center for Research in Computing Technology
Cambridge, MA 02138

## 1 Introduction

There has been extensive recent research in finding robust and efficient numerical representations for geometric objects. Ideally, one would like to use the field of real numbers with the operations of addition, subtraction, multiplication and division and taking roots of polynomial equations. The standard floating-point representation seems to be a reasonable approximation to real arithmetic, as it is fast, universally available, and well supported by programming languages [3]. However, on any computer architecture, the set of representable floating-point numbers is a finite subset of the set of real numbers, and the output of any floating-point operation must be rounded to a representable number. Geometric algorithms are especially sensitive to the effects of round-off error because the behavior of an algorithm is determined by the signs of expressions on the inputs. If the actual value of an expression is near zero, even a small amount of round-off error can cause its computed value to have the wrong sign. It is possible that the signs computed for a collection of expressions do not correspond to any realizable geometric object. When this happens, the floating-point implementation of an algorithm will fail in some unpredictable manner. It is generally possible to create geometric algorithms that always generate a realizable output by enforcing consistency among the signs [13] [1]. Even so, only for a few geometric constructions have methods and proofs been given that the resulting objects have reasonably small numerical errors [9] [10] [4] [5].

Assuming integral or rational inputs, it is always possible to implement a geometric algorithm so that it only needs to test the signs of polynomial expressions with *integer* coefficients and variables. For example, if the geometric objects being represented are "linear" (lines or planes), then one can use exact rational arithmetic, and the sign of a rational number is determined by the signs of its integer numerator and denominator. For non-linear domains, such as quadrics or spline curves and surfaces, there are techniques involving resultants that can reduce any computation to the sign of an integer determinant. Even in the "linear" case, the size of the integers can grow quite large. In fact, we know of no way of avoiding exponential growth in the size of the numerator and denominators for a sequence of operations on polyhedral solids: union/intersection, translation/rotation, convex hulls. In general, exact arithmetic is not an efficient alternative to floating-point arithmetic, even though it solves the problems of robustness.

It has been observed that most of the time, the floating-point expression has the correct sign and there is no need to use exact arithmetic. Fortune and Van Wyk [6] have developed LN, a system for efficient exact arithmetic based on this principle. The challenge is determining when exact arithmetic is not needed. At compile time, LN derives error bounds for each expression and outputs C++ code. At run-time, this code first evaluates the expressions in floating-point. If the magnitude of the resulting value is greater than the error bound, it returns the resulting sign. If not, the code then uses the full precision required for an exact evaluation. It appears that in practice, LN generates programs that are more efficient than systems that use many levels of accuracy [8].

In this paper we describe an experimental application of LN to the task of implementing a geometric algorithm. We first implemented a naive floating-point algorithm for taking the union and intersection of polyhedral solids and then translated it into LN. This algorithm was chosen because it is relatively small but requires a sufficient amount of computation to test the practicality of LN. The translation exceeded the precompiled limits of the system because of the sheer number of expressions required, even though the algorithm is relatively simple.[1] Fortune and Van Wyk indicate that implementations using exact arithmetic require a different philosophy from floating-point implementations, so it was not surprising that the direct translation "overloaded" the system [2]. We next made some changes to the primitives,

*Supported by NSF grants NSF-CCR-91-157993 and NSF-CCR-90-09272

[1] We estimate the result without the limit to be about 24,000 lines of C++ to evaluate three geometric primitives, which we considered to be impractically large.

reducing the number of expressions required and their degrees such that the code could be compiled (result was 8671 lines of C++).

Most of the required changes illustrate that using *any* exact arithmetic package efficiently is difficult because it involves a different and nontrivial kind of thought from that required when using floating-point arithmetic. For example, it is sometimes necessary to replace a single evaluation of a high degree expression with several evaluations of lower degree expressions. However, there were certain design choices in the current version of LN which made translation especially difficult, specifically, the need to treat each type of object separately. A geometric point can be one of many types: a vertex of the input polygon, the intersection of an edge of one polygon and a face of another, the intersection of two line segments. Any expression acting on points, such as one needed to test whether four points are coplanar, would have to be replicated to include a version for each possible combination of input types. This results in a number of expressions exponential in the number of types. Currently, there is no way of automatically generating all these expressions. Also, the goal of reducing the number of types (and thus the number of expressions) is sometimes in conflict with the goal of reducing the degrees of the expressions.

Our suggestion is to use dynamic evaluation of the error bounds, which the developers of LN had previously considered and rejected because of the higher estimated cost of computation at run time. Using the dynamic approach allows the use of the C++ inheritance and class derivation mechanisms to simplify the code in terms of its length as well as the amount of expertise required in its design. Although this approach sacrifices running time, it eliminates the combinatorial explosion in the number of expressions and other difficulties that we encountered. Partly as a result of our experiments, Fortune and Van Wyk are currently creating a version of LN with dynamic evaluation to see whether the computational demands are, in fact, too high.

The naive union/intersection algorithm is presented in Section 2; Section 3 discusses the implementation of the algorithm using LN; and Section 4 discusses the issues that arose and describes the alternate approach of using dynamic evaluation.

## 2 The Algorithm

We first implemented an algorithm for taking the union and intersection of polyhedra using floating-point arithmetic. The implementation is naive in that it treats floating-point arithmetic as if it were valid rational arithmetic, ignoring any problems that might arise from rounding.

In our experiments, the polyhedra are represented by the coordinates of their vertices rather than by the equations of the planes of the faces. We chose the vertex representation for a number of reasons; for example, the plane representation risks large errors when rotating, especially at points far away from the center of rotation [7]. The vertices have integer homogeneous coordinates, and the faces of each input polyhedron must form a *face lattice* in three dimensions: a set of triangles such that any two triangles are either disjoint or share either a vertex or an entire edge in common [11]. Any non-triangular polygonal face can be represented as a set of coplanar triangular faces. Representing faces as sets of triangles is sufficient to illustrate numerical issues; in practice, more sophisticated data structures and algorithms would be used. The algorithm returns the union or intersection of the two polyhedra.

### 2.1 Accommodating the Polyhedra

First, the polyhedra have to be "accommodated," that is, vertices and edges need to be added such that the union of the vertices and edges of the two polyhedra together form a face lattice. For example, the unaccommodated polyhedra in Figure 1 might look like Figure 2 after accommodation. Suppose we are accommodating polyhedron $P$ with respect to polyhedron $Q$. For each face $ABC$ of $P$, we find the set of intersection of each face of $Q$ with the plane of $ABC$. These edges are found by first computing, for each face $DEF$ of $Q$, to which side of the plane of $ABC$ $D$, $E$, and $F$ lie. The position of $D$ with respect to $ABC$ is determined by the sign of a four by four integer determinant (Section 3.1). From this information, it is possible to find the edges of intersection. For example, if all three vertices of $DEF$ are found to be in the plane of $ABC$, the set of edges would be all three edges of $DEF$.

Suppose the edges of $DEF$ intersect the plane of $ABC$ in two points. These are the endpoints of the segment of intersection of $DEF$ with respect to the plane of $ABC$. Switching the roles of $ABC$ with $DEF$ gives the endpoints of a collinear segment. The four endpoints can be ordered linearly with respect to any of the x, y, and z coordinates to deduce the intersection of the two segments, which is the intersection of the two faces. If an endpoint of the intersection is on an edge of $ABC$, a new vertex is added there, and the edge of the triangle split at that point. We now have a set of segments which may look like that in Figure 3.

If any edge of $DEF$ lies in the plane of $ABC$, we need to find the portions of that edge that lie inside the face $ABC$, adding new vertices and splitting the edge as needed. The intersection of two line segments is found by parameterizing the lines determined by the two segments. We only want those segments inside the face; to determine if a segment is in the face, we check whether its midpoint is in the face using cross products. The result would be as in Figure 4.

Next, additional segments need to be added such that each closed region inside the face is a triangle as in Figure 5, and thus the accommodated polygon is once again a face lattice. Segments are added as long as they do not intersect any previously existing or added segments.

### 2.2 Finding the Union/Intersection

Since we have accommodated the polyhedra such that their faces together form a face lattice, each face $f$ of $P$ will be a proper subset of 1) the boundary, 2) the exterior, or 3) the interior of $Q$. For (1), if $f$ equals

$g \in Q$ with the *same* normal direction, then a single copy of $f$ and $g$ appears in (the boundaries of) $P \cap Q$ and $P \cup Q$; otherwise, neither appear. If (2), $f$ appears in $P \cup Q$. If (3), $f$ appears in $P \cap Q$. To test if $f$ lies inside $Q$, we project a ray $r$ from the centroid of $f$ in some random direction. For each face $g$ of $Q$, the intersection point $v$ of $r$ with the plane of $g$ is computed, and then $v$ is tested if it lies inside $g$. The number of faces pieced by $r$ is odd if $f$ lies inside $Q$ and even if outside. The same process is applied to each face of $Q$ with respect to $P$.

## 3 LN

Steven Fortune and Christopher Van Wyk have developed a programming language called LN, designed to provide efficient exact arithmetic for computational geometry. LN is based on the fact that many geometric primitives are actually the evaluation of signs of polynomial expressions [6], which can always be written as polynomials in the input coordinates. This is evident if the inputs are all "original" vertices. Suppose, however, that at least one of the input vertices to a primitive is derived, say, as the intersection of an edge and a plane. The homogeneous coordinates of computed vertices can themselves be expressed as polynomial expressions in the original inputs and then substituted into the expression for the geometric primitive. All the variables in the resulting expression are now coordinates of original vertices.

LN generates C++ code which uses floating-point arithmetic to evaluate expressions and which uses exact arithmetic only when necessary. At compile time, a maximum error bound is computed for each expression from the maximum error that could be incurred from rounding each variable in the expression. At run time, the floating-point values of each variable are first used to evaluate the expression, and the magnitude of the result is compared to the precomputed error bound for that expression. If the computed magnitude of the expression is greater than that of the error bound, the computed sign must be correct and no further computation is necessary. It is only when the magnitude of the expression evaluated in floating-point is less than the error bound that exact evaluation of the expression is required. An implementation using LN can be about as fast as a "naive" floating-point implementation and it can be even faster than a robust floating-point implementation because it does not have to use tolerancing and other techniques to achieve reliability.

### 3.1 Translation into LN

To implement an algorithm using LN, it is necessary to know the possible types of the inputs for each expression. LN will treat each combination of input types separately even if the expression itself is the same. For example, a half-plane test with all original vertices will have a different error bound than one in which all three vertices are computed, and these cases will be treated differently. We need to find how many different types of vertices are involved in the union/intersection algorithm, and which expressions they could be inputs to.

An accommodated polyhedron can have three different types of vertices: original input vertices (type 0), vertices derived as the intersection of a segment and a plane (type 1), and vertices derived as the intersection of two line segments, whose endpoints are all original (type 2). The coordinates of all three types can be expressed as polynomials in the coordinates of the points from which they were derived.

Clearly, each original coordinate is a degree 1 polynomial. To compute the intersection of a segment $AB$ and a plane $CDE$, parameterize segment $AB$ as $sA + tB$, and find (any) $s$, $t$ such that $\det(sA + tB, C, D, E) = 0$. Solving, we find that s and t are fourth degree polynomials, and since type 1 vertices have the form $sA + tB$, they are of degree 5. Type 2 vertices are derived in a similar fashion; they are of degree 4. We also need to compute the centroids of faces and the intersection of rays and faces. We find that there are 10 different types of each.

We next determined which of these points could be inputs to which expressions, and tabulated the results:

| **PRIMITIVE** | **# EXPS** | **POSSIBLE INPUTS** |
|---|---|---|
| Point-in-plane | 1 | type 0 |
| Vertex comparison | 6 | type 0, 1, 2 |
| Half-plane | 10 | type 0, 1, 2 |
| | 10 | 10 centroids, type 0 |
| | 10 | 10 ray casts, type 0 |
| **TOTAL** | **37** | |

The first 10 half-plane tests come from the 10 ways to input the three different types of vertices. (Note that inputting vertices of types 0, 1, and 1 is considered to be the same as inputting vertices of types 1, 0 and 1.) The second 10 come from testing to see whether a centroid is in a face for the degenerate case of two coplanar faces. Because the face being tested will always have three original vertices, there are only 10 cases, one for each type of centroid. The third 10 come from testing to see whether the intersection of a random ray and a face lies in the face, but again the face always has three original vertices.

LN was "overloaded" when it tried to compile all of the code required for the 37 expressions and the separate constructors for the 23 different types of points. It could only compile about two-thirds of the LN code, but had already written about 16,000 lines of C++. There was evidently much more code required than the implementors of LN had anticipated, even for a small example such as this. Clearly, we had to change the set of expressions we used to evaluate the primitives and find ones which were optimal for exact evaluation, particularly in LN.

#### 3.1.1 Optimizations

The first optimization had already been implemented in the accommodation of the polyhedra. Instead of testing whether a segment is inside a face by computing the midpoint of the segment and doing a point-in-face test, we conclude that a segment is in a face if both of its endpoints are in the face, and thus we avoid computing an extra type of vertex. The midpoint was calculated in the floating-point version to simplify the algorithm. The use of auxiliary points often eliminates consideration of degenerate cases; however, it is not suitable for exact arithmetic because of the resulting higher degree expressions, and it is particularly unsuitable for LN because it introduces a new type of vertex.

The next change was suggested by Dr. Fortune. Instead of computing type 2 vertices as the intersections of two segments, these points are now computed in the same way as type 1 vertices. A fifth point $E$ not in the plane of the intersecting segments $AB$ and $CD$ is chosen at random, and the former type 2 vertex is computed as the intersection of segment $AB$ and plane $CDE$. This eliminates type 2 vertices, reducing the total number of types of vertices to 2. Now there are only 4 types of faces, and any function that takes 3 vertices as inputs has only 4 possible combinations of inputs instead of 10. Note that this change is specific to LN and is not optimal for exact arithmetic in general; the vertices converted from type 2 to type 1 are now of higher degree (5 instead of 4), and all the expressions involving these vertices are larger than necessary from the point of view of exact evaluation.

Another alteration is based on the observation that it is unnecessary to compute the actual intersection of a random ray and the faces it is being projected onto. Instead, it suffices to check the orientations of certain groups of four vertices. The ray $r$ projected out of centroid $c$ hits the plane determined by the face $ABC$ if $\det(cABC)$ and $\det(rABC)$ have different signs, and it passes through the face $ABC$ if $\det(rcAB)$, $\det(rcBC)$, and $\det(rcCA)$ all have the same sign. This technique avoids computing the high degree point $v$ where the ray $r$ hits the plane $ABC$, and it avoids computing a point-in-face test with $v$, a computation that evaluates expressions of degree 21. This change is not specific to LN, and it illustrates the difference between floating-point and exact arithmetic and the level of difficulty involved in finding efficient ways to evaluate geometric primitives.

With these optimizations, the total number of constructors needed was reduced from 23 to 13, and the total number of expressions needed was reduced from 37 to 21. The code was now reduced to a sufficiently small size such that it could be compiled.

These optimizations were by no means the only possible changes that could have been made. For example, consider determining the segment of intersection of two faces. This involves finding the linear ordering of four collinear type 1 vertices. The implemented exact algorithm accomplishes this by using a fifth coplanar point which is not collinear with the four points being ordered (in the algorithm, a type 0 vertex can always be found which fulfills these criteria), and half-plane tests are performed with this fifth point and pairs of the collinear points, revealing the relative positions of the four points along the line. As these half-plane tests involve two type 1 vertices (degree 5) and one type 0 vertex (degree 1), the half-plane expressions are of degree 11. However, given a point $p$ which is the intersection of segment $AB$ and the plane of $CDE$, we are essentially trying to determine whether $p$ is in the face $CDE$. This can be equivalently done by testing the orientations of four tetrahedra, $ABDE$, $ABEF$, and $ABFD$. They will all be of the same sign if and only if $p$ lies inside $DEF$. Notice that the degree of these expressions is only 4 since all vertices involved are type 0. Again, this example shows the consideration on the part of the user required to use exact arithmetic efficiently.

These examples dramatically illustrate the difficulties in converting to exact arithmetic. There is no systematic or mechanical way of reducing the degrees of expressions or eliminating them entirely, nor is there a way to methodically reduce the number of vertices that need to be computed. We also see that in some cases the current design of LN requires extra changes, some of which are not optimal for exact arithmetic in general.

## 4 Using LN: Issues

The above example of finding the union and intersection of two polyhedra is a relatively small one; what happens when the application is even more complex than this?

### 4.1 Specific Issues

**I**. For a three-input orientation primitive, having two vertex types 0 and 1 happens to be convenient because regardless of the order of the vertices in the code, it is always possible to rotate them to fit one of the four combinations of 000, 001, 011, and 111 without changing their orientation. In general, things are not so simple; Polya-Burnside enumeration indicates a minimal set of combinations of input types that a primitive must handle, and a whole layer must be added in the program on top of the expression evaluation to convert any ordering of the inputs to one of these combinations. It quickly becomes difficult to do all the bookkeeping necessary to keep track of all the different cases.

**II**. There is a tradeoff between minimizing the degrees of expressions and minimizing the number of types and hence the number of expressions. For example, consider the case where two triangles intersect only in one point which lies on an edge of each triangle. This point can be computed as the intersection of these two edges to yield a vertex whose coordinates are degree 4, as opposed to finding this point as the intersection of a line and a plane, of degree 5. To avoid introducing a new type and many new expressions in LN, it is necessary to use the computation with higher degree.

**III**. There is the tradeoff between smaller degreed expressions and the number of such expressions that need to be evaluated. The example in Section 3.1.1 demonstrated that the ordering of the four collinear points can be done by either evaluating 12 degree 4 expressions or 5 degree 11 expressions. Currently, LN forces the choice of many smaller degree expressions, which may not be the optimal choice in general.

**IV**. Consider the cases where there are a large number of degeneracies; i.e. when the value of many of the geometric expressions is zero. For each of these cases, the magnitude of the floating-point computation will always be smaller than that of the maximum error bound, requiring an exact computation to verify that the expression really does equal zero. When modeling the real world, there are likely to be many of these degenerate cases; for example, objects lie in the plane of the floor, or of a tabletop, and walls, ceilings, and floors are parallel or perpendicular to each other. A system like LN may be forced to do exact computations more often than statistics derived from a uniform distribution of inputs might suggest.

**V**. Like any reasonable exact arithmetic system, LN limits the input values to some specified bit length. The output of our polyhedral union/intersection algorithm could not be used as input to the same code, since the explicit bit lengths of some of the output vertices will be significantly larger than the input vertices; type 1 vertices will be about 5 times as big. We cannot expect to avoid this growth in bit length by keeping the vertex locations implicit through introduction of new types. Even if LN allowed dynamic definition of types, the number of resulting expressions would grow exponentially. One needs to try to find "safe" ways of rounding the output coordinates back to small integers in a way that either preserves the combinatorial structure or modifies it in a reasonable manner. Milenkovic [11][12] proposes such a rounding algorithm, but an efficient algorithm is yet to be found.

Issues I, II, and III indicate ways that the current design of LN increases the programmer effort over that generally required for implementation using exact arithmetic. These difficulties might be eliminated through the use of dynamic error evaluation as described in the next section. Issue IV is faced by all exact arithmetic systems that attempt to reduce cost by using floating-point whenever it is sufficiently accurate. It remains to be seen how much impact it will have in practice. Issue V is faced by all exact arithmetic systems, and geometric rounding is one possible solution.

### 4.2 Dynamic Evaluation

The main bottleneck in using LN is having to treat each type of vertex separately. Suppose that instead of static evaluation of error bounds at compile time, it used dynamic evaluation at run time. This would allow the programmer to use a single C++ base class vertex from which all the different "types" of vertices are derived. Thus, all of the expressions which formerly had a different instance for each combination of types of inputs would only have one instance because the types of the inputs would all be of a single type "vertex." At run time, when an expression is instantiated, C++ virtual functions in the vertex class would provide the following values for each coordinate: its floating-point approximation, an upper bound on its magnitude, an estimate of the number of bits of error, and (if an exact computation is necessary) its exact value. Virtual functions provide the ability to reference these values for each particular vertex without explicit knowledge of the vertex type.

Using this dynamic approach saves much coding time but sacrifices running time; however, for large applications, a small increase in running time would certainly be an acceptable tradeoff for the large decrease in programming effort and output code size (which also might adversely effect running time through excessive page faults). In fact, any naive floating-point algorithm such as the one described in Section 2 could be translated almost directly into a dynamic evaluation scheme, although it might not be as efficient as a well thought out exact implementation.

## 5 Conclusion

There seems to be a conflict between robustness and practical efficiency of geometric algorithms. The language LN, while achieving its goal of increasing efficiency of exact arithmetic at run time, may turn out to be impractical when the sheer size of the code and the time spent in designing and implementing such a large program is taken into account. A possible improvement is to use dynamic evaluation, which would save most of the time and effort that is required to program using the current LN package. It is both time consuming and intellectually demanding to keep track of the types of all the computed quantities, to enumerate the different functions for each of the different possible combinations of input types, and to make clever optimizations to reduce the number of types. These demands are in addition to those ordinarily required to use exact arithmetic: to simplify and minimize the degrees of expressions. The dynamic approach would sacrifice a small factor in running time but when considered on the whole, is certainly the more acceptable alternative. We propose that LN be modified to test this paradigm.

## References


[1] T. K. Dey and C. L. Bajaj and K. Sugihara. "On good triangulations in three dimensions" *Internat. J. Comput. Geom. Appl.* 2:1 (1992), pp. 75–95.

[2] S.Fortune, C.Van Wyk. "LN User Manual." AT&T Bell Laboratories, 1993.

[3] S. Fortune. From *Directions in Geometric Computing*, Information Geometers, publishers, 1992.

[4] Steven Fortune. Stable Maintenance of Point-Set Triangulations in Two Dimensions. In *30th Annual Symposium on the Foundations of Computer Science*, IEEE, October 1989.

[5] S. Fortune and V. J. Milenkovic. Numerical Stability of Algorithms for Line Arrangements. *Seventh Annual ACM Symposium on Computational Geometry*, North Conway, N.H., June 10-12, 1991, pp. 334–341.

[6] S. Fortune, C. Van Wyk. "Efficient Exact Arithmetic for Computational Geometry." *Proceedings of the Symposium on Computational Geometry*, ACM, 1993.

[7] C. M. Hoffmann. *Geometric and Solid Modeling*, Morgan Kaufmann, publishers, 1989.

[8] M. Karasick and D. Lieber and L. R. Nackman. Efficient Delaunay Triangulation using Rational Arithmetic. *ACM Transactions on Graphics*, 10:71–91, January 1991.

[9] Victor J. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.* Technical Report CMU-CS-88-168, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, July 1988.

[10] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the Symposium on Computational Geometry*, ACM, 1989.

[11] V. Milenkovic. "Rounding Face Lattices in $d$ Dimensions." *Proceedings of the Second Canadian Conference on Computational Geometry*, Jorge Urrutia, Ed., University of Ottawa, Ontario, August 6-10, 1990, pp. 40–45.

[12] V. Milenkovic. "Rounding Face Lattices in the Plane." *First Canadian Conference on Computational Geometry*, Montreal, Quebec, Canada, 1989.

[13] K. Sugihara. *Construction of the Voronoi diagram for one million generators in single-precision arithmetic.* Report 89-05, Dept. Math. Engrg., Univ. Tokyo, Tokyo, Japan, 1989.