Efficient Delaunay Triangulation Using Rational Arithmetic

MICHAEL KARASICK, DEREK LIEBER, and LEE R. NACKMAN
IBM Thomas J. Watson Research Center

Many fundamental tests performed by geometric algorithms can be formulated in terms of finding the sign of a determinant. When these tests are implemented using fixed-precision arithmetic such as floating point, they can produce incorrect answers; when they are implemented using arbitrary-precision arithmetic, they are expensive to compute. We present adaptive-precision algorithms for finding the signs of determinants of matrices with integer and rational elements. These algorithms were developed and tested by integrating them into the Guibas–Stolfi Delaunay triangulation algorithm. Through a combination of algorithm design and careful engineering of the implementation, the resulting program can triangulate a set of random rational points in the unit circle only four to five times slower than can a floating-point implementation of the algorithm. The algorithms, engineering process, and software tools developed are described.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software—*efficiency, reliability, and robustness*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*geometric algorithms, languages, and systems*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design (CAD)*

General Terms: Algorithms, Design, Experimentation, Languages, Performance, Reliability

Additional Key Words and Phrases: Delaunay triangulation, rational arithmetic, robust geometric computation, triangulation

1. INTRODUCTION

Lack of robustness is a serious practical problem in finite-precision implementations of geometric algorithms. It is folklore that most solid modeling systems fail on some problems of practical interest [7]. The same is true even for implementations of much simpler geometric algorithms. Further, the use of ad hoc approaches to achieving robustness, often based on "epsilon" tolerances, dramatically increases the complexity of practical implementations of geometric algorithms. The problems are twofold: geometric degeneracies and numerical error.

It is very difficult to ensure that an implementation deals with all of the special cases caused by geometric degeneracies. Numerical error is caused by the substitution of floating point for real numbers. Because of the complexity of many

Authors' address: Manufacturing Research Department, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0730-0301/91/0100-0071 $01.50

geometric algorithms, it is difficult to obtain symbolic bounds (as distinguished from running error analyses) on numerical error that are tight enough to be useful [6]. Moreover, numerical error is often magnified "near" geometric degeneracies, precisely when the most accuracy is needed. Indeed, techniques for avoiding geometric degeneracies through conceptual perturbation of the input data [5, 28] require exact computation.

The growing recognition that numerical error is a serious impediment to practical, efficient, and robust implementations of geometric algorithms has led to increased attention to the problem [3, 4, 8–10, 12, 14, 17–19, 21, 23]. Recent surveys appear in [6] and [13]. There appears to be a fundamental dichotomy in the approaches taken between the use of exact computation over restricted domains (e.g., rational and algebraic numbers) and the use of fixed-precision approximations to real numbers (e.g., floating point).

Exact computation provides simplicity and assured robustness at the expense of computational efficiency. It provides simplicity in the sense that algorithms formulated over the restricted domain map directly to implementations, without need to treat numerical error. Moreover, the handling of geometric degeneracies is vastly simplified by the absence of complex interactions between numerical error and tests for degeneracy. However, exact computation must be implemented using arbitrary-precision arithmetic, implying that each operation takes time that is a function of operand sizes. Consequently, algorithms that minimize the number of arithmetic operations do not necessarily minimize running time.

Sugihara [26] obtains constant-time arithmetic operations by bounding the precision of input data that, in turn, fixes the precision needed for exact computation. In particular, he implements set operations on polyhedra represented as combinations of plane equations (numerical data) and adjacency relationships among vertices, edges, and faces (combinatorial data). Given bounds on the precision of the plane equation coefficients, it is straightforward to compute a priori bounds on the precision needed throughout. Although this approach limits the cost of exact computation, restrictions on input precision introduce difficult new problems, as described in [13].

Alternatively, fixed-precision arithmetic can be used together with techniques that consistently resolve ambiguities arising from numerical error. Geometric algorithms generally use data structures that contain numerical data and combinatorial data. Problems arise because of inconsistencies between the two. Karasick [14], in his work on robust intersection of polyhedra, resolves the inconsistencies by altering the numerical data to be consistent with the combinatorial data. Unfortunately, some of these alterations may have global effects or, in the case of degeneracies, may not be possible. Nevertheless, Karasick's polyhedral intersection program is very robust in practice. Milenkovic [17, 18], in his work on robust line arrangement algorithms, resolves ambiguities by altering both the numerical and combinatorial data to ensure that point/line incidence tests can be done reliably in fixed-precision arithmetic. His *data normalization* approach shifts vertices that are too close and shifts vertices and "cracks" edges when an edge and a vertex are too close. In the worst case, data normalization can be computationally expensive and can make arbitrarily large changes in the geometry. Milenkovic's *hidden variable* approach approximates

the line arrangement problem with a related curve arrangement problem that allows reliable incidence tests in fixed-precision arithmetic. Fortune [8] describes an algorithm that triangulates a point set using finite-precision arithmetic, and Sugihara and Iri [27] describe an algorithm that constructs a Voronoi diagram using finite-precision arithmetic. Both algorithms guarantee topological consistency even in the presence of unreliable numerical computations. Guibas et al. [10] describe the construction of robust algorithms using "epsilon predicates," relational predicates that incorporate numerical uncertainty. This approach leads to elegant algorithms, but at the expense of efficiency.

Most primitive tests used in geometric algorithms can be formulated as a function that maps input parameters to a small number of discrete values. In particular, many such primitive tests, including orientation of $d + 1$ points in E^d and point-hyperplane classification, can be formulated as the sign of the determinant of a matrix. Since computing values of determinants is very expensive in arbitrary-precision arithmetic, it is natural to ask whether it is possible to compute the sign without computing the value. We are not aware of any general way to do this. However, given a matrix A, we might be able to find $\text{sign}(|A|)$ by transforming A into another matrix A', finding $\text{sign}(|A'|)$, and then using the properties of the transformation to map $\text{sign}(|A'|)$ into the sign of $|A|$. This strategy is effective if the two mappings, $A \rightarrow A'$ and $\text{sign}(|A'|) \rightarrow \text{sign}(|A|)$, can be done more efficiently than finding $\text{sign}(|A|)$ directly.

Below, we develop several variations on this theme, all of which approximate matrix elements by intervals with low-precision endpoints. For low enough precision, the savings in arithmetic cost can far outweigh the additional cost of doing interval arithmetic. The simplest approach takes a matrix A with integer elements and transforms it into a matrix A' whose elements are integer intervals, and then uses interval arithmetic to find $|A'|$. For example,

$$|A| = \begin{vmatrix} -191{,}285{,}375{,}129{,}284{,}278{,}128 & -193{,}294{,}274{,}204{,}273{,}238{,}012 \\ 242{,}474{,}290{,}147{,}147{,}023{,}937 & 234{,}294{,}274{,}294{,}973{,}293{,}384 \end{vmatrix}$$

has the same sign as

$$|A'| = \begin{vmatrix} [-192,-191] & [-194,-193] \\ [242,243] & [234,234] \end{vmatrix}.$$

If the interval $|A'|$ does not span zero, then $\text{sign}(|A|)$ is $\text{sign}(|A'|)$; otherwise more precision is necessary. In the worst case the matrix is singular, and no advantage is to be had by using less than full precision.

In practice the worst case might not occur very often, and these transformation approaches might yield a large savings. To test this hypothesis, we replaced floating-point arithmetic with an existing arbitrary-precision rational-arithmetic package in the Guibas–Stolfi Delaunay triangulation algorithm [11]. The rational implementation was more than 10,000 times slower than the floating-point one. By a combination of engineering and application of the above ideas, we were able to reduce this factor to about four or five for random points in the unit circle.

In particular, we developed procedures that take a matrix with arbitrary-precision integer or rational elements and adaptively compute the sign of the matrix determinant. Since our objective was to obtain a reliable and efficient

implementation of the Guibas–Stolfi algorithm, we were guided by repeated experimentation: At each stage, we measured its performance on a simple example, and then either modified the sign-of-determinant procedure or engineered the code. As the running time decreased, we used larger and larger examples, varying both the number of input points and their precision (number of bits in their coordinates).

The weakness of this approach is that the algorithm and code improvements are not independent, and so it is difficult to draw quantitative conclusions about the value of a particular improvement. For example, improvements in a greatest common divisor procedure might greatly reduce the running time of rational arithmetic computations. However, such an improvement might later be obviated by reformulating the computations. This demonstrates that early experiments can be inconclusive. However, they are necessary to identify and eliminate the program "hot spots" that typically dominate the running time of an implemented algorithm [2].

The remainder of this paper describes the sign-of-determinant procedures, the experiments performed, and the software techniques that facilitated the experimentation.

2. BACKGROUND

2.1 The Guibas–Stolfi Delaunay Triangulation Algorithm

The *Delaunay diagram* of a set S of points in the plane is the subdivision of the plane induced by those directed line segments (a,b) that connect points of S and have either of the following properties:

(1) (a, b) is an edge of the convex hull of S; or
(2) for every point $c \in S$ and $d \in S$ on the left and right of (a, b), respectively, the oriented circle abc does not contain d in its interior

If no four points of S are cocircular, then the Delaunay diagram is a triangulation, called the *Delaunay triangulation* of S. Otherwise, the Delaunay diagram of S can be transformed into a (nonunique) Delaunay triangulation by adding edges to the diagram.

Guibas and Stolfi [11] describe an elegant recursive algorithm that merges Delaunay triangulations of linearly separated point sets L and R into a Delaunay triangulation of $L \cup R$. This merge is done by repeatedly deleting edges from the triangulations of L and R and then creating a new edge that connects a point of L to a point of R. The algorithm uses two simple geometric tests. The *InCircle* predicate,

$$\begin{vmatrix} 1 & x_a & y_a & x_a^2 + y_a^2 \\ 1 & x_b & y_b & x_b^2 + y_b^2 \\ 1 & x_c & y_c & x_c^2 + y_c^2 \\ 1 & x_d & y_d & x_d^2 + y_d^2 \end{vmatrix} < 0,$$

is true if and only if property 2 above holds for four points a, b, c, and d. The CCW predicate,

$$\begin{vmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{vmatrix} > 0,$$

is true if and only if the points a, b, and c have a counterclockwise orientation. The Guibas–Stolfi algorithm does $O(n \log n)$ InCircle and CCW tests to triangulate n points.

As a starting point, we began with a floating-point implementation of the Guibas–Stolfi algorithm and directly replaced the floating-point arithmetic with rational arithmetic. Rational numbers were implemented as pairs of arbitrary-precision integers, and results of arithmetic operations were simplified using the binary GCD algorithm [15]. Arbitrary-precision integers were represented as arrays of radix-2^{16} digits. We chose this radix because the implementation language, C^{++}, uses 32-bit arithmetic and does not detect integer overflow. Arithmetic operations were implemented using classical algorithms [15], assignment was implemented by copying, and the digit arrays were allocated and reclaimed using the C^{++}-provided storage management procedures. The CCW and InCircle tests used by the Guibas–Stolfi algorithm were implemented by finding the sign of the appropriate determinant using cofactor expansion down the 1's column.

Our first experiment was to triangulate ten random points in the unit circle. For double-precision points, the implementation using floating-point arithmetic took 0.1 seconds; for rational points with comparable precision (2-digit numerator and 3-digit denominator), the implementation using arbitrary-precision rational arithmetic took 1200 seconds, generating intermediate values with as many as 81 digits.

Even though we expected the rational implementation to be slower than the floating-point implementation, we were surprised by the size of the disparity. We concluded that we had to reduce both the number of operations and the cost of individual operations. We began by investigating better ways to compute the sign of the determinant of a matrix with arbitrary-precision integer elements.

2.2 Simulation Model of Computation

To accurately predict the performance of an algorithm implemented using exact integer arithmetic, the model of computation must capture the behavior of the implementation. Several models of computation are described in [1]. The simplest, which we call the *unit-cost* model, allows arithmetic and comparison operations on arbitrary-precision integers in constant time. In the *log-cost* model, an integer a is manipulated as a bit string of length

$$l(a) = \begin{cases} \lfloor \lg |a| \rfloor + 1, & a \neq 0 \\ 1, & a = 0, \end{cases}$$

and arithmetic operations take time that is a function of this size. However, on most computers data is manipulated in word-sized "chunks," or *digits*. Therefore,

we use a *simulation* model of computation in which integers are manipulated as radix-r numbers with

$$l(a) = \begin{cases} \lfloor \log_r |a| \rfloor + 1, & a \neq 0 \\ 1, & a = 0 \end{cases}$$

digits.

To determine the cost of arithmetic operations in the simulation model, we assume that they are implemented using the classical algorithms described in [15].[1] If a and b are integers with α and β digits, respectively, then addition, subtraction, and comparison take $O(\alpha + \beta)$ time, multiplication takes $O(\alpha\beta)$ time, and division takes $O(\alpha\beta - \beta^2)$ time.

3. USING INTEGER INTERVALS

Since we are not aware of any general way to compute the sign of a determinant without first computing its value, we take the approach discussed in Section 1: A matrix A is transformed into a matrix A' for which $|A'|$ is often faster to compute than $|A|$, and then properties of the transformation are used to obtain sign($|A|$). Specifically, a matrix of rationals is first transformed into a matrix of integer intervals with low-precision endpoints, and then the determinant is computed. If the resulting interval $|A'|$ does not include zero, then sign($|A|$) is known; otherwise, higher precision endpoints are used, and the process is repeated. Regardless, it is necessary to find efficient means to compute the determinant of a matrix of integers.

The determinant $|A|$ of an $n \times n$ matrix A can be found efficiently using *cofactor expansion* [20] if n is small. A variant on classical Gaussian elimination, *Sgauss*, finds the sign of the determinant using only integer arithmetic. We first present the *Sgauss* algorithm and then compare *Sgauss* with cofactor expansion using the simulation model.

Classical Gaussian elimination replaces element a_{ij} of matrix A with

$$a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j},$$

which can be rewritten as

$$\frac{1}{a_{11}} \begin{vmatrix} a_{11} & a_{1j} \\ a_{i1} & a_{ij} \end{vmatrix}.$$

If A is an $n \times n$ matrix, then after the first elimination step (involving a_{11}) the first row of A remains unchanged, the $n - 1$ elements below a_{11} are zero, and the remaining $(n-1)^2$ elements form a submatix of intergers divided by a_{11}. For

[1] We use the classical algorithms because they are simple to implement and more efficient for relatively small operands than the more sophisticated algorithms described in [1] and [15].

example, if A is a 4×4 matrix, then this first elimination step yields

$$\frac{1}{a_{11}^2}\left|\begin{matrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{14} \\ a_{21} & a_{24} \end{vmatrix} \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{14} \\ a_{31} & a_{34} \end{vmatrix} \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{41} & a_{42} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{41} & a_{43} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{14} \\ a_{41} & a_{44} \end{vmatrix} \end{matrix}\right|.$$

The second elimination step reduces A to a 2×2 matrix, and produces the factor

$$\frac{1}{\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}}.$$

In general, if A is an $n \times n$ matrix, then *Sgauss* transforms $|A|$ into a product of $n - 2$ factors and the determinant of a 2×2 matrix S. The sign of $|A|$ is found by examining the sign of $|S|$ and the signs of the factors with odd powers. Note that unlike standard Gaussian elimination, *Sgauss* need not do any division because only the sign of $|A|$ is desired. A simple analysis shows that given an $n \times n$ integer matrix with δ-digit elements, *Sgauss* uses $O(\delta^2 4^n)$ time. A more careful analysis is possible using the following idea: The ith *Sgauss* elimination step transforms the determinant of an $n \times n$ matrix into the product of a scalar $D(i)$ raised to the power $n - 1 - i$ and the determinant of an $(n - 1) \times (n - 1)$ matrix. It is easy to see that

$$D(n-1) = |A| \prod_{i=1}^{n-2} D^{n-1-i}(i).$$

Using Hadamard's inequality to bound $|A|$ in the above recurrence relation, we can bound $D(n)$. This in turn is used to bound the number of single-digit operations used by *Sgauss*. The resulting analysis is shown graphically in Figure 1. *Sgauss* becomes significantly more efficient than cofactor expansion for $n \geq 5$.

Since *Sgauss* is only marginally less efficient than cofactor expansion for $n =$ 3, 4, and because we want to generalize the procedures to larger matrices, we used only *Sgauss* for the CCW and InCircle tests in subsequent experiments.

Unfortunately, even *Sgauss* with exact computation is far too inefficient for practical implementation of the Guibas–Stolfi algorithm. By approximating the elements of the CCW and InCircle matrices using integer intervals (with low-precision endpoints), the running time can often be reduced. In fact, this can be done adaptively: if the determinant computed with interval arithmetic spans zero, the computation is repeated with higher precision interval endpoints, continuing until the sign is obtained. This remains an expensive computation if the determinant is zero. This algorithm is implemented by Program $Sgauss_{interval}$, shown in Figure 2.

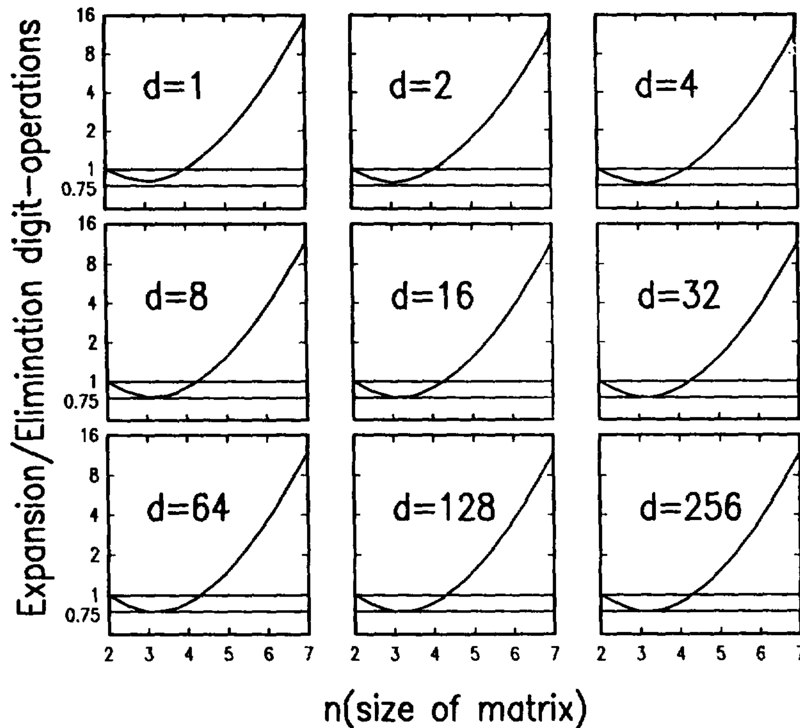

Fig. 1. Theoretical number of single-digit operations required by cofactor expansion and *Sgauss*. The data plotted above are ratios of the number of single-digit operations for *Sgauss* and cofactor expansion, varying matrix size (n) and matrix-element size (d). For $n = 2$, the two algorithms are identical; for $n = 3, 4$, cofactor expansion is slightly better; and for $n \geq 5$, *Sgauss* is better. The data are obtained using Hadamard's inequality to bound the determinant magnitudes.

Using $Sgauss_{\text{interval}}$ reduced the time required to triangulate ten random points from 1200 to 97 seconds. $Sgauss_{\text{interval}}$ reduced the time by an order of magnitude because it often found the sign on the first iteration: the 96 CCW tests and 31 InCircle tests required only 176 interval determinants to be computed by $Sgauss_{\text{interval}}$. We concluded that the approach is effective, but that the running time was still unacceptably high (i.e., 1000 times slower than floating point). We therefore decided to try to reduce the running time by engineering the implementation.

3.1 Engineering the Implementation

Several straightforward changes to the implementation of rational arithmetic reduced the running time of the triangulation algorithm from 97 to 75 seconds:

(1) Digit-shift operators were implemented and used to speed the boxing operation in step 3 of $Sgauss_{\text{interval}}$.

(2) Operations like $a = a + 1$ were replaced with compound-assignment operators.

(1) Transform the rational matrix into integer matrix, e.g.,

$$\begin{vmatrix} \frac{a_{11}}{b_{11}} & \frac{a_{12}}{b_{12}} & \frac{a_{13}}{b_{13}} \\ \frac{a_{21}}{b_{21}} & \frac{a_{22}}{b_{22}} & \frac{a_{23}}{b_{23}} \\ \frac{a_{31}}{b_{31}} & \frac{a_{32}}{b_{32}} & \frac{a_{33}}{b_{33}} \end{vmatrix} = \frac{\begin{vmatrix} a_{11}b_{12}b_{13} & b_{11}a_{12}b_{13} & b_{11}b_{12}a_{13} \\ a_{21}b_{22}b_{23} & b_{21}a_{22}b_{23} & b_{21}a_{22}b_{23} \\ a_{31}b_{32}b_{33} & b_{31}a_{32}b_{33} & b_{31}b_{32}a_{33} \end{vmatrix}}{b_{11}b_{12}b_{13}b_{21}b_{22}b_{23}b_{31}b_{32}b_{33}}$$

and discard the denominators, which are positive.

(2) Let h be the number of digits in the matrix element with smallest magnitude.

(3) "Box" each element with a low-precision integer interval, as follows:

   (a) Let $k = r^{h-1}$, where r is the radix.

   (b) Drop the $h - 1$ least significant digits of each element a_{ij} by replacing a_{ij} with the interval $[\lfloor a_{ij}/k \rfloor, \lceil a_{ij}/k \rceil]$.

(4) Compute the sign of the determinant using *Sgauss* with integer-interval arithmetic.

(5) If this sign is $[-1, 1]$, then repeat steps 3 and 4 with a smaller h.

Fig. 2. Program $Sgauss_{\text{interval}}$. Given a matrix with rational elements, compute the sign of its determinant adaptively.

(3) Rational number comparison, originally performed by examining the sign of the difference, was reimplemented to try special case tests (sign and size comparison) first, and then to compute the sign of the numerator of the difference. This improvement was important because rational number comparison is used in sorting the points, the first step in the Guibas–Stolfi algorithm.

Statistics gathered by the rational-arithmetic package indicated that simplifying rational numbers after every arithmetic operation rarely reduced the numerator and denominator sizes by more than one digit. Consequently, we dispensed with simplification altogether. This reduced the running time from 75 to 30 seconds. The size of the largest intermediate value increased from 108 to 110 digits. Initially, we believed that simplification would be worthwhile because two random integers have a nontrivial greatest common divisor about 40 percent of the time (Theorem 4.5.2D, [15]). However, in the simulation model of computation, there is no effect unless the greatest common divisor is at least a digit in size. We found, at least for our experiment, that this was not the case very often.

Nevertheless, we were surprised by the large size of the intermediate results generated. When we investigated, we found that some InCircle determinants were zero. This was surprising given the random input data.

3.2 A Superfluous Numerical Test in the Guibas–Stolfi Algorithm

Close inspection of the Guibas–Stolfi algorithm reveals a boundary condition for which a combinatorial test can detect a zero InCircle determinant, thus avoiding the expense of computing a zero determinant. This boundary condition appears

in the merge loop in Figure 23 of [11]:

```
WHILE InCircle
  [basel.Dest, basel.Org, lcand.Dest, lcand.Onext.Dest]
           DO . . . OD
```

It is possible that basel.Org and lcand.Onext.Dest are the same vertex.[2] This happens if there are only two edges incident to vertex basel.Dest. Since the InCircle test returns *false* when the determinant is zero, the code can be modified to check for this case:

```
DO
  IF basel.Org = lcand.Onext.Dest
    THEN EXIT FI;
  IF NOT InCircle[basel.Dest, basel.Org, lcand.Dest, lcand.Onext.Dest]
    THEN EXIT FI;
  . . .
OD;
```

Similarly, the symmetric case can be modified to read:

```
DO
  IF basel.Dest = rcand.Oprev.Dest
    THEN EXIT FI;
  IF NOT InCircle[basel.Dest, basel.Org, rcand.Dest, rcand.Oprev.Dest]
    THEN EXIT FI;
  . . .
OD;
```

These changes reduced the number of InCircle tests from 31 to 28, which decreased the running time from 30 to 20 seconds and the size of the largest intermediate value from 110 to 20 digits. Note that evaluating zero determinants was very expensive.

3.3 Storage Management

The improvements described thus far have focussed on reducing the number of machine arithmetic operations required. With these improvements in place, we observed that 70 percent of the running time was spent executing the C storage management functions, *malloc* and *free*. In our experiment, storage was allocated and reclaimed for approximately 120,000 integers, of which 90,000 had at most ten digits.

Therefore, we augmented the integer data structure to contain preallocated space for 31 digits. We also added a pointer to the data structure that points to either the preallocated storage or to storage obtained from *malloc* for integers exceeding 31 digits. (This self reference complicated the assignment operation.) Running time decreased from 20 to 6 seconds, effectively eliminating storage management overhead for our experiment.

[2] It is interesting to ask why floating-point implementations of the unmodified algorithm do not encounter numerical error in this situation. If either cofactor expansion or Gaussian elimination is used to compute the determinant, it is not difficult to see that zero will be produced even with floating-point arithmetic. However, if the determinant is evaluated as a polynomial, the result will depend on the order in which the terms are summed.

Further improvement was obtained by implementing the assignment operation to copy using the largest possible "single-instruction chunks" available on our machine (32 bits). Previously, copying had been done one byte at a time. This reduced the running time from 6 to 5 seconds.

3.4 Reformulating the InCircle and CCW Determinants

The first columns of the CCW and InCircle determinants are all 1s. This fact can be exploited to further reduce the running time by performing the first Gaussian elimination step directly on the rational matrix, immediately reducing the effective size of the matrix by one, but substantially increasing the cost of converting from a rational to an integer matrix.

Another way to exploit these columns of 1s is to convert from an integer to a rational matrix by multiplying down columns instead of along rows (as in Program $Sgauss_{\text{interval}}$ of Figure 2). Then, the first Gaussian elimination step can be done efficiently using integer arithmetic. When this is done, the $x^2 + y^2$ column of the InCircle matrix has much larger integers than the other columns. For our experiment, the resulting x and y columns could have at most 11-digit elements, but the $x^2 + y^2$ column could have 68-digit elements. This disparity can be eliminated by boxing the columns independently, which does not change the sign of the determinant.

When we integrated this modification of $Sgauss_{\text{interval}}$ into the Guibas–Stolfi algorithm, the running time decreased from 5 to 3.5 seconds even though the size of the largest intermediate value increased from 20 to 45 digits.

3.5 Incremental Computation of the Determinant of a Matrix

If rational numbers are represented as ratios of *scaled integers*, $I = m \times r^e$, where m is an arbitrary-precision integer, r is the radix, and e is an exponent, then the elements of the CCW and InCircle matrices can be scaled independently by the boxing step of program $Sgauss_{\text{interval}}$. For example, the positive integer $a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0$ can be bounded in the interval $[a_n, a_n + 1] \times r^n$, thus requiring only one digit to approximate each element.

For our experiment, this did not yield any improvement because the elements in each column of the CCW and InCircle determinants had approximately the same size. Therefore, the extra overhead of constructing and normalizing the scalar integer representation just increased the running time.

Often, only the high-order digit or two of the integer matrix elements need be examined when finding the sign of the determinant. If arithmetic could be carried out most-significant digits first, the entire computation could be terminated when the sign becomes available. In particular, it might not be necessary to complete the conversion of the rational matrix into an integer matrix.

This can be done by representing rational numbers as ratios of scaled integers and using a simple incremental algorithm for the conversion. Using this approach, the Guibas–Stolfi algorithm runs six times slower than $Sgauss_{\text{interval}}$ on our test data. The overhead of maintaining the state of the incremental multiplier is far too high unless the elements have a huge number of digits.

Another approach is to use Preparata and Vuillemin's [22] algorithm for computing the product of two integers in left-to-right digit order. Their algorithm, which uses a redundant digit representation, obtains the most significant digit of the product after examining each digit of the two integers. (This digit may be zero even though the product is nonzero). We did not investigate this approach, and it would be an interesting topic to pursue. However, the next section describes a way to entirely avoid the rational- to integer-matrix conversion.

4. USING INTEGER-INTERVAL RATIOS

Thus far, we have been converting a rational matrix to an integer matrix before finding the sign of its determinant. The multiplications required can be a significant part of the total time required to find the sign of the determinant. If the conversion cost can be reduced, the total time might be reduced.

Recall that the primitive *Sgauss* elimination step replaces each element a_{ij} of a matrix A with

$$\frac{1}{a_{11}} \begin{vmatrix} a_{11} & a_{1j} \\ a_{i1} & a_{ij} \end{vmatrix}.$$

If each element of A is a rational number a_{ij}/b_{ij}, then the above elimination step replaces it with

$$\frac{b_{11}}{a_{11}} \begin{vmatrix} \frac{a_{11}}{b_{11}} & \frac{a_{1j}}{b_{1j}} \\ \frac{a_{i1}}{b_{i1}} & \frac{a_{ij}}{b_{ij}} \end{vmatrix} = \frac{\begin{vmatrix} a_{11}b_{i1} & a_{1j}b_{ij} \\ a_{i1}b_{11} & a_{ij}b_{1j} \end{vmatrix}}{a_{11}b_{i1}b_{1j}b_{ij}}.$$

If A is of size n, then after the first elimination step (using a_{11}/b_{11}), the first row of A remains unchanged, the $n - 1$ elements below a_{11}/b_{11} are zero, and the remaining $(n - 1)^2$ elements form a submatrix of size $n - 1$. If A is a 4×4 matrix, then the first elimination step produces

$$\begin{vmatrix} \frac{a_{11}}{b_{11}} & \frac{a_{12}}{b_{12}} & \frac{a_{13}}{b_{13}} & \frac{a_{14}}{b_{14}} \\ 0 & \frac{\begin{vmatrix} a_{11}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{22}b_{12} \end{vmatrix}}{a_{11}b_{21}b_{12}b_{22}} & \frac{\begin{vmatrix} a_{11}b_{21} & a_{13}b_{23} \\ a_{21}b_{11} & a_{23}b_{13} \end{vmatrix}}{a_{11}b_{21}b_{13}b_{23}} & \frac{\begin{vmatrix} a_{11}b_{21} & a_{14}b_{24} \\ a_{21}b_{11} & a_{24}b_{14} \end{vmatrix}}{a_{11}b_{21}b_{14}b_{24}} \\ 0 & \frac{\begin{vmatrix} a_{11}b_{31} & a_{12}b_{32} \\ a_{31}b_{11} & a_{32}b_{12} \end{vmatrix}}{a_{11}b_{31}b_{12}b_{32}} & \frac{\begin{vmatrix} a_{11}b_{31} & a_{13}b_{33} \\ a_{31}b_{11} & a_{33}b_{13} \end{vmatrix}}{a_{11}b_{31}b_{13}b_{33}} & \frac{\begin{vmatrix} a_{11}b_{31} & a_{14}b_{34} \\ a_{31}b_{11} & a_{34}b_{14} \end{vmatrix}}{a_{11}b_{31}b_{14}b_{34}} \\ 0 & \frac{\begin{vmatrix} a_{11}b_{41} & a_{12}b_{42} \\ a_{41}b_{11} & a_{42}b_{12} \end{vmatrix}}{a_{11}b_{41}b_{12}b_{42}} & \frac{\begin{vmatrix} a_{11}b_{41} & a_{13}b_{43} \\ a_{41}b_{11} & a_{43}b_{13} \end{vmatrix}}{a_{11}b_{41}b_{13}b_{43}} & \frac{\begin{vmatrix} a_{11}b_{41} & a_{14}b_{44} \\ a_{41}b_{11} & a_{44}b_{14} \end{vmatrix}}{a_{11}b_{41}b_{14}b_{44}} \end{vmatrix}$$

In general, the result of an elimination step can be simplified by the following sign-preserving transformations:

(1) Drop the positive common denominator b_{1j} from every element of column j.
(2) Drop the positive common denominator b_{i1} from every element of row i.
(3) Factor out a_{11} from the first column and $1/a_{11}$ from the others, yielding factor $1/a_{11}{}^{n-2}$.

After applying these transformations to the preceding example, we obtain

$$\frac{1}{a_{11}^2}\begin{vmatrix} \dfrac{\begin{vmatrix} a_{11}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{22}b_{12} \end{vmatrix}}{b_{22}} & \dfrac{\begin{vmatrix} a_{11}b_{21} & a_{13}b_{23} \\ a_{21}b_{11} & a_{23}b_{13} \end{vmatrix}}{b_{23}} & \dfrac{\begin{vmatrix} a_{11}b_{21} & a_{14}b_{24} \\ a_{21}b_{11} & a_{24}b_{14} \end{vmatrix}}{b_{24}} \\ \dfrac{\begin{vmatrix} a_{11}b_{31} & a_{12}b_{32} \\ a_{31}b_{11} & a_{32}b_{12} \end{vmatrix}}{b_{32}} & \dfrac{\begin{vmatrix} a_{11}b_{31} & a_{13}b_{33} \\ a_{31}b_{11} & a_{33}b_{13} \end{vmatrix}}{b_{33}} & \dfrac{\begin{vmatrix} a_{11}b_{31} & a_{14}b_{34} \\ a_{31}b_{11} & a_{34}b_{14} \end{vmatrix}}{b_{34}} \\ \dfrac{\begin{vmatrix} a_{11}b_{41} & a_{12}b_{42} \\ a_{41}b_{11} & a_{42}b_{12} \end{vmatrix}}{b_{42}} & \dfrac{\begin{vmatrix} a_{11}b_{41} & a_{13}b_{43} \\ a_{41}b_{11} & a_{43}b_{13} \end{vmatrix}}{b_{43}} & \dfrac{\begin{vmatrix} a_{11}b_{41} & a_{14}b_{44} \\ a_{41}b_{11} & a_{44}b_{14} \end{vmatrix}}{b_{44}} \end{vmatrix}.$$

The entire process can then be repeated, reducing the size of the matrix each time, until a 2×2 matrix is obtained. In general, this elimination procedure, which we call $Sgauss_{\text{rational}}$, transforms the determinant of a rational matrix A of size n into a product of $n - 2$ factors and the determinant of a 2×2 matrix S. The sign of $|A|$ is found by examining the sign of $|S|$ and the signs of the factors having odd exponents. Given an $n \times n$ matrix with rational elements that have ν-digit numerators and δ-digit denominators, $Sgauss_{\text{rational}}$ uses $O((\nu + \delta)^2 4^n)$ single-digit arithmetic operations.

A more useful result is obtained by comparing the number of operations required by $Sgauss_{\text{rational}}$ to the number required by an algorithm that first converts the rational matrix to an integer matrix and then uses *Sgauss*. Using Hadamard's inequality to bound the sizes of the determinants, we obtain the data in Figure 3. These plots show that the total time required is almost independent of d. Thus, the total time required depends both on n and the amount of precision necessary. If $Sgauss_{\text{rational}}$ evaluates the determinant to full precision, then it is better for matrices of size $n \leq 8$; on the other hand, as the precision needed decreases, $Sgauss_{\text{rational}}$ becomes more attractive. This can be exploited by an adaptive-precision version of $Sgauss_{\text{rational}}$, obtained by observing that a rational number can be "boxed" by a ratio of integer intervals. For example, the rational number

$$\frac{-284{,}283{,}293{,}293{,}002{,}348}{209{,}078{,}384{,}027{,}273{,}234}$$

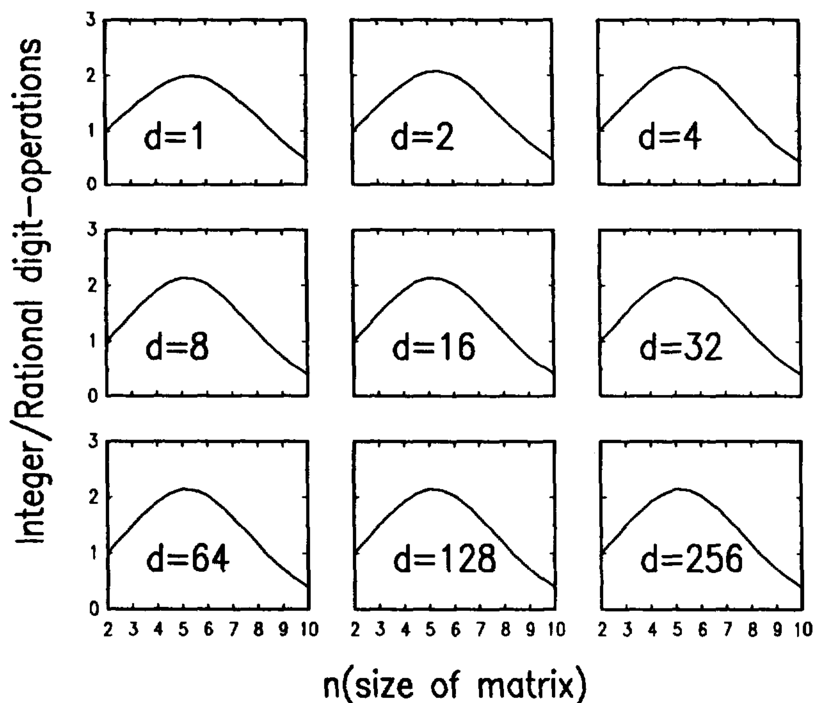

Fig. 3. Theoretical number of operations required by $Sgauss_{\text{rational}}$. The data plotted compare the time required by $Sgauss_{\text{rational}}$ to compute the sign of the determinant of a rational matrix with the time required to convert the rational matrix to an integer matrix, and then use *Sgauss* to compute the sign of the determinant. Matrix elements have d-digit numerators and denominators.

is contained in the ratio of intervals

$$\frac{[-285, -284]}{[209, 210]}.$$

Program $Sgauss_{\text{interval-ratio}}$ (see Figure 4) implements these ideas. It increased the running time of the Guibas–Stolfi algorithm on our test set from 3.5 to 5.6 seconds. The size of the largest intermediate value decreased from 45 to 38 digits.

For our experiment, the test points have numerators and denominators with few digits, and so the cost of converting from a rational to an integer matrix is cheap enough that $Sgauss_{\text{interval}}$ is still more efficient than $Sgauss_{\text{interval-ratio}}$. If the numerators and denominators have more digits, the time required to convert from a rational to an integer matrix will begin to dominate the computation time of $Sgauss_{\text{interval}}$; conversely, we do not expect the computation time required by $Sgauss_{\text{interval-ratio}}$ to increase significantly. This conjecture is supported by the data of Figure 5, which measures the time required to triangulate fixed-size point sets whose coordinates have numerators and denominators with successively larger magnitudes. Both $Sgauss_{\text{interval-ratio}}$ and $Sgauss_{\text{interval}}$ usually converge on the first iteration, and so the times in Figure 5 dominate the time required by $Sgauss_{\text{interval}}$ to convert from a rational to an integer matrix.

(1) Obtain the interval-ratio matrix A by boxing each element of the rational matrix using an integer-interval ratio. (If a numerator has ν digits and its corresponding denominator has δ digits, then drop the low order $min(\nu,\delta) - 1$ digits from each.)
(2) Subtract the first row of A from the other rows of A, simplify the result by factoring, and drop the first row and column.
(3) Find the sign of $|A|$ using $Sgauss_{rational}$ with integer-interval arithmetic.
(4) If sign($|A|$) is $[-1,1]$, then repeat steps 1 to 3 with additional precision.

Fig. 4. Program $Sgauss_{interval\text{-}ratio}$. Given an InCircle or CCW matrix A with rational elements, compute the sign of its determinant adaptively.

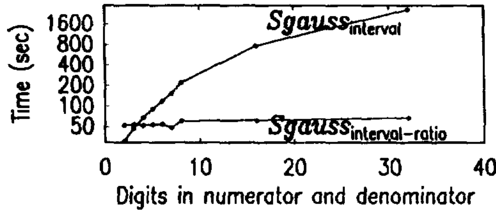

Fig. 5. Times required to triangulate 50 points using $Sgauss_{interval}$ and $Sgauss_{interval\text{-}ratio}$. For random points in the unit circle, $Sgauss_{interval}$ is more efficient than $Sgauss_{interval\text{-}ratio}$ if each numerator and denominator has fewer than four digits. As the magnitudes of the numerators and denominators increase, the time used by $Sgauss_{interval}$ to convert from rational CCW and InCircle matrices to integer matrices begins to dominate the computation.

4.1 Continued Fractions

Another way to approximate rational elements using lower precision is to use simple continued fraction expansions. Given a rational number α and an integer Q, there is an algorithm that finds the best approximation to α with denominator no greater than Q [16]. In fact, if $q \leq Q$ is the denominator in the best approximation, the error is no greater than $1/(qQ)$. Despite the low error, the cost of computing the approximation outweighs the benefit. Furthermore, the denominators of the matrix elements are potentially different from one another. Thus, the sign of the matrix determinant must be computed using rational arithmetic. In the next section, a more efficient approximation technique is developed.

5. A FIXED-PRECISION-INTERVAL FILTER

When all of the improvements we have discussed were integrated into the Guibas-Stolfi algorithm, it became fast enough that we could use a larger test case. The new test data consisted of 500 random points in the unit circle, computed initially in double precision. For input to the rational implementation, these were converted to their rational equivalents. We compared the floating-point implementation to our fastest rational implementation, the one that uses $Sgauss_{interval}$. The floating-point implementation took 14 seconds, while the rational implementation took 470 seconds. Although this ratio was three orders of magnitude better than our initial baseline experiment, the rational implementation remained impractically slow.

Measurements revealed that the adaptive algorithms often found the sign of the CCW and InCircle determinants on the first iteration using single-digit

(16-bit) approximations of the elements. Nevertheless, these calculations incurred the large expense of arbitrary-precision arithmetic. To avoid this expense, we implemented a fixed-precision "filter" that uses machine arithmetic directly on low-precision approximations. If this succeeds, the determinant's sign is found very quickly. If not, the adaptive algorithms continue until the sign is found.

Since the objective was a fast filter, we used a simple approximation. A rational number a/b can be approximated by the interval $[p/q, (p + 1)/q]$, where q is positive and p is computed as follows:

(1) Choose denominator q; and set $p = aq/b$, $r = aq \bmod b$.
(2) If r is zero, then form the approximation $[p, p]/q$ and exit.
(3) Otherwise, if r is negative, then subtract 1 from p.
(4) Finally, form the approximation, $[p, p + 1]/q$.

If q is chosen to be a power of two, then the multiplication can be done by shifting. If the same value of q is used to approximate every element of a matrix, then q can be factored out of the matrix and $Sgauss_{\text{interval}}$ used directly to find the sign. Using 32-bit (hardware) integer arithmetic, a CCW test with 15-bit elements and an InCircle test with 6-bit elements can be implemented in this way.

With our 500-point test set, these approximations find the sign in 13,272 of the 13,277 CCW tests, but in only 65 of the 6,780 InCircle tests. We therefore decided that a 32-bit fixed precision filter was ineffective. However, 53-bit integers can be manipulated using the 53-bit mantissas provided by our machine's (IEEE standard) floating-point hardware. In addition, somewhat higher precision element approximations can be used if integer overflow is detected by examining the floating-point exponent after each arithmetic operation. If examining the exponent indicates that overflow has occurred, the interval is widened.

These ideas were incorporated in a new procedure, $Sgauss_{\text{double-interval}}$, which finds the sign of the determinant of a matrix with double-precision interval elements. We used $Sgauss_{\text{double-interval}}$ in the Guibas–Stolfi algorithm as a fixed-precision filter and with $q = 2^{24}$. For points in the unit circle, this allows CCW tests to be done without overflow and InCircle tests to be done with overflow possible only in the last elimination step. If $Sgauss_{\text{double-interval}}$ failed to find the determinant's sign, then $Sgauss_{\text{interval}}$ was used, starting with 2-digit (32-bit) integer-interval endpoints and doubling the number of digits used in successive iterations until the sign was found. Using this approach, the running time for our 500-point test case decreased from 470 to 316 seconds.

5.1 Caching Frequently Computed Values

Double-precision intervals are computed each time $Sgauss_{\text{double-interval}}$ is invoked. However, these interval approximations do not change between invocations because they are associated with the coordinates of the point set to be triangulated. Thus it makes sense to retain the approximations so that they need not be recomputed. Since we approximate a rational number by

$$\frac{[n, n + 1]}{2^d},$$

we cache n (a scaled integer represented using double precision) with the rational number that it approximates. This trivial optimization reduced the running time from 316 to 195 seconds. Likewise, caching with each point the $x^2 + y^2$ used in the InCircle test reduced the running time from 195 to 59 seconds.

6. SOFTWARE

The experimental approach that we adopted was feasible only because we had appropriate software tools. In particular, it was essential to be able to easily define arithmetic operations on various domains. For example, once we had matrix operations and interval arithmetic, it was trivial to obtain matrices with interval elements. Those same software techniques yield efficient, modular, and maintainable programs. Because of their importance to this work, we describe the tools in this section.

6.1 Mapping Arithmetic Domains into C++ Classes

Our software was written in C++ [24], a language that allows efficient manipulation of low-level machine entities like bits and words, but also allows the construction of abstract data types (*classes*) and hierarchies thereof. C++ provides a concise notation for initializing objects, copying them, displaying them, and manipulating them with functions and operators. Together with a compiler-enforced discipline for creating objects as they "enter scope" and destroying them when they "leave scope," these facilities permit algorithms to be written clearly, debugged easily, and changed quickly (thereby encouraging experimentation) while retaining most of the execution efficiency of a more primitive programming language such as C.

We used the C++ class mechanism to construct and manipulate numbers from different arithmetic domains and to define a set of functions and operators common across all these domains. We implemented the following arithmetic classes:

int the class of numbers representable as 32-bit two's complement integers,

integer the class of numbers representable as 53-bit signed magnitude integers,

Integer the class of unlimited precision integers,

double the class of numbers represented as 53-bit mantissas scaled by 11-bit exponents,

Rational the class of numbers represented as ratios of Integers,

Interval the class of numbers represented as intervals with int-, integer-, Integer-, double-, and Rational-endpoints,

Ratio the class of numbers represented as ratios of Intervals of Integers,

Matrix the class of square matrices with elements of type Integer, double, Rational, Interval of integer, Interval of double, Interval of Integer, Interval of Rational, and Ratio of Interval of Integer

Many versions of the *Interval, Ratio,* and *Matrix* classes could be implemented; the above list shows some of the ones we actually used.

The following example shows a code fragment, which implements two of the addition operators for rational numbers.

```
Rational operator + (const Rational& x, const Rational& y)
  {
  return Rational(x.num * y.den + x.den * y.num, x.den * y.den);
  }
Rational operator + (const Integer& x, const Rational& y)
  {
  return Rational(x * y.den + y.num, y.den);
  }
```

We actually defined five different versions of the "+" operator, corresponding to possible operand pairings that might be encountered. C^{++} automatically invokes the correct one, depending on the situation at hand. For example, in the code fragment:

```
Integer a(1);
Rational b(1,2);
Rational c = a + b;
```

C^{++} would automatically invoke the function "operator + (Integer, Rational)."

6.2 Class Templates

Frequently, the same source code could describe several classes, perhaps with additions or omissions of a few lines of code. For example, the source code for any matrix class is independent of the matrix element type, and the source code for any interval class is independent of the type of interval endpoints.

We wanted to write such source code once, as a *template* parameterized by one or more types, and use it to automatically generate object code for the desired types. The C^{++} language currently lacks such a template mechanism (but see [25]), so we implemented it straightforwardly using the UNIX® macro processor, *m4*. Consider the following (incomplete) template for *Ratio* classes:

```
// Generic Ratio Arithmetic
GENERIC_def_C(Ratio,'Den','Num')
# include "Ratio(Num,Den).h"
// Sum.
Ratio(Num,Den) operator+(const Ratio(Num,Den)& x, const Ratio(Num,Den)& y)
  {
  return Ratio(Num,Den)(x.num * y.den + x.den * y.num, x.den * y.den);
  }
```

The template corresponds closely to a regular C^{++} class definition, except that the type of the numerator and denominator is parameterized. The **GENERIC_def_C** directive tells our template preprocessor the "base" name of the template and the names by which the template is to be parameterized. For example, invoking the template preprocessor with "Num = int" and "Den = int" would

® UNIX is a trademark of AT&T Bell Laboratories.

generate a file named "Ratio_int_int.C" containing:

```
# include "Ratio_int_int.h"
// Sum.
Ratio_int_int operator + (const Ratio_int_int& x, const Ratio_int_int& y)
  {
  return Ratio_int_int(x.num * y.den + x.den * y.num, x.den * y.den);
  }
```

7. CONCLUSIONS AND FUTURE WORK

A combination of algorithm improvements and engineering have reduced the running time of a rational arithmetic implementation of the Guibas–Stolfi Delaunay triangulation algorithm by more than three orders of magnitude. Our current program is able to correctly triangulate a set of 500 points in 59 seconds; a floating-point implementation requires 14 seconds, with no guarantee of correctness. Thus we have obtained the robustness guaranteed by rational arithmetic, paying for this with a five-fold increase in running time.

In the course of refining and testing our code, we gained several insights about implementing rational arithmetic and using it in the Guibas–Stolfi algorithm:

(1) It is rarely necessary to examine every bit of every element of a rational or integer matrix to compute the sign of its determinant. But when it is necessary (i.e., when the determinant is zero), the computation is very expensive. Therefore, it is important to replace degenerate determinant calculations with combinatorial tests whenever possible. For example, modifying the Guibas–Stolfi algorithm to avoid calculating certain zero determinants significantly reduced the running time.

(2) Simplifying rational numbers after arithmetic operations does not pay because it rarely reduces the size of a number by more than one digit.

(3) Memory management can be a major cost; it seems worthwhile to increase memory usage to simplify and lower the cost of memory management. For example, preallocating a fixed amount of storage sufficient to hold small integers is effective.

(4) Arbitrary-precision arithmetic is very expensive, so it is worthwhile to use memory for caching wherever possible.

(5) If hardware support is available for arithmetic on sufficiently large integers, a fixed-precision filter used to quickly cull cases that can be evaluated in low precision is very effective.

Appropriate software tools are essential to the experimentation that drives algorithm development and engineering. By using an object-oriented programming language like C++, we were able to write our algorithms in a clear and natural notation, using standard arithmetic operators and functions. By defining a standard set of operators and functions, we were able to switch between different arithmetic domains by simply changing a few `# define` or `# include` lines in the C++ source and recompiling. Class templates allowed us during the course of our experimentation to quickly and easily create many different versions of classes like *Interval*, *Ratio*, and *Matrix*.

Although many important geometric tests can be formulated in terms of the sign of a determinant, it is not yet clear how well the techniques described in this paper will work on other algorithms or with nonrandom data. However, we are heartened by the fact that our experience in tuning this algorithm mirrors the techniques for writing efficient programs described in [2]. Also, it is likely that data encountered in practical applications of Delaunay triangulation will include some input that leads to zero determinants. If the proportion of such determinants remains small, the techniques we have described are likely to remain effective. Otherwise, it might become necessary to devise a fast test for a zero determinant. One possibility would be to use modular arithmetic [15] to compute the determinant. Since sign determination (other than zero) is expensive in modular arithmetic, it might best be used to cull zero determinants after using the fixed-precision filter described previously.

ACKNOWLEDGMENTS

We thank Michael O'Connor for suggesting the use of modular arithmetic and one of the referees for bringing the report by Preparata and Vuillemin [22] to our attention.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BENTLEY, J. L. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
3. BERN, M. W., KARLOFF, H. J., RAGHAVAN, P., AND SCHEIBER, B. Fast geometric approximation techniques and geometric embedding problems. In *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry* (June 1989). ACM, New York, 1989, 292–301.
4. DOBKIN, D. P., AND SILVER, D. Recipes for geometry and numerical analysis—Part I: An empirical study. In *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry* (June 1988). ACM, New York, 1988, 93–105.
5. EDELSBRUNNER, H., AND MÜCKE, E. P. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry* (June 1988). ACM, New York, 1988, 118–133.
6. FAROUKI, R. T. Numerical stability in geometric algorithms and representations. In *Proceedings Mathematics of Surfaces III,* D. C. Hanscomb, Ed. Oxford University Press, New York, 1989.
7. FORREST, A. R. Computational geometry and software engineering: Towards a geometric computing environment. In *Techniques for Computer Graphics,* R. A. Earnshaw and D. F. Rogers, Eds. Springer Verlag, New York, 1987, 23–37.
8. FORTUNE, S. J. Stable maintenance of point-set triangulations in two dimensions. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science* (Oct. 1989). IEEE, New York, 1989.
9. GREENE, D. H., AND YAO, F. F. Finite-resolution computational geometry. In *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science* (1986). IEEE, New York, 1986, 143–152.
10. GUIBAS, L., SALESIN, D., AND STOLFI, J. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry* (June 1989). ACM, New York, 1989, 208–217.
11. GUIBAS, L. AND STOLFI, J. Primitives for the manipulation of general subdivisions and the computations of Voronoi diagrams. *ACM Trans. Graph. 5,* 2 (Apr. 1985), 74–123.
12. HOFFMANN, C., HOPCROFT, J., AND KARASICK, M. Towards implementing robust geometric computations. In *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry* (June 1988). ACM, New York, 1988, 106–117.

13. HOFFMANN, C. M. The problem of accuracy and robustness in geometric computation. *IEEE Comput. 22* (1989), 31–42.
14. KARASICK, M. On the representation and manipulation of rigid solids, Ph.D. thesis, McGill Univ., Montreal, 1988.
15. KNUTH, D. E. *Seminumerical Algorithms. Vol. 2. The Art of Computer Programming.* Addison-Wesley, Reading, Mass., 1973.
16. LOVÁSZ, L. *An Algorithmic Theory of Numbers, Graphs and Convexity. Vol. 50. CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia, 1986.
17. MILENKOVIC, V. J. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artif. Intell. 37* (Dec. 1988), 377–401.
18. MILENKOVIC, V. J. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry* (June 1989). ACM, New York, 1989, 197–207.
19. MUDUR, S. P., AND KOPARKAR, P. A. Interval methods for processing geometric objects. *IEEE Comput. Graph. Appl. 4*, 2 (Feb. 1984), 7–17.
20. NERING, E. D. *Linear Algebra and Matrix Theory.* John Wiley, New York, 1970.
21. OTTMANN, T., THIEMT, G., AND ULLRICH, C. Numerical stability of geometric algorithms. In *Proceedings of the Third Annual ACM Symposium on Computational Geometry* (June 1987). ACM, New York, 1987, 119–125.
22. PREPARATA, F. P., AND VUILLEMIN, J. E. Practical cellular dividers. INRIA Tech. Rep. 807, Rocquencourt, France, March 1988.
23. SEGAL, M., AND SÉQUIN, C. Partitioning polyhedral objects into nonintersecting parts. *IEEE Comput. Graph. Appl. 8*, 1 (Jan. 1988), 53–67.
24. STROUSTRUP, B. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1987.
25. STROUSTRUP, B. Parameterized types for C++. In *Proceedings of the USENIX C++ Conference* (Oct. 1988). USENIX Association.
26. SUGIHARA, K. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci. 39* (1989), 236–247.
27. SUGIHARA, K., AND IRI, M. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. Paper presented at the First Canadian Conference on Computational Geometry. Montreal, Aug. 1989.
28. YAP, C. A geometric consistency theorem for a symbolic perturbation scheme. In *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry* (June 1988). ACM, New York, 1988, 134–142.

Received March 1989; revised December 1989; accepted January 1990

Editor: L. Guibas