Algorithmica (1997) 17: 111–132

Algorithmica

Evaluating Signs of Determinants Using Single-Precision Arithmetic[1]

F. Avnaim,[2] J.-D. Boissonnat,[2] O. Devillers,[2] F. P. Preparata,[3] and M. Yvinec[4]

Abstract. We propose a method of evaluating signs of 2×2 and 3×3 determinants with b-bit integer entries using only b and $(b+1)$-bit arithmetic, respectively. This algorithm has numerous applications in geometric computation and provides a general and practical approach to robustness. The algorithm has been implemented and compared with other exact computation methods.

Key Words. Computational geometry, Exact arithmetic, Precision, Robust algorithms.

1. Introduction. Most decisions in geometric algorithms are based on signs of determinants. For example, deciding if a point belongs to a given half-space or a given ball reduces to evaluating the sign of a determinant. Moreover, such evaluations are often the only numerical parts of the entire algorithm. Therefore, it is crucial to have reliable answers to such tests.

This observation relates to the very model of Computational Geometry, whose assumption is that geometric parameters are real numbers and that arithmetic operations are performed with infinite precision. Obviously, such assumption does not hold when algorithms are translated into computer programs, where the parameters are represented either as integers or as floating-point numbers. Floating-point implementations, although naively a reasonable approach to real-number arithmetic, have been shown to have serious shortcomings, since they may cause not only numerical errors but also fatal membership errors (such as inclusion of a point in an interval to which it does not belong, etc.).

This difficulty has received some deserved attention in recent years (see, e.g., [For1], [GY], [HHK], [Mil1], [Mil2], [Mil3], [GSS], and [SI]) and several approaches have been proposed on how to obviate the shortcoming. The common objective is to produce *robust* algorithms, namely algorithms whose answer is a (small) perturbation of the correct answer (as produced by the infinite-precision algorithm). As noted by Fortune [For1], there are basically two categories of approaches to this objective: The most common one resorts to approximate (i.e., rounded) computations, and uses properties of the assumed

---

[1] This work was partially supported by the ESPRIT Basic Research Action 7141 (AL-COMII) and by NSF Grant CCR 91-96176. Part of this work was done while J.-D. Boissonnat was visiting Brown University.
[2] INRIA, BP93, 06902 Sophia-Antipolis cedex, France. Jean-Daniel.Boissonnat@sophia.inria.fr. Olivier.Devillers@sophia.inria.fr.
[3] Brown University, Department of Computer Science, Providence, RI 02912-1910, USA. franco@cs.brown.edu.
[4] INRIA and CNRS, URA 1376, Laboratoire I3S, 250 Rue Albert Einstein, 06560 Valbonne, France. Mariette.Yvinec@sophia.inria.fr.

Received September 12, 1994; revised March 5, 1995. Communicated by K. Mehlhorn.

primitives to establish the topological correctness of the results (i.e., robustness) (see, e.g., [For1], [For2], [FM], [HHK], and [GSS]). The other uses exact (i.e., integer) computations but, since multiprecision integer arithmetic is required (d-fold for a dimension d determinant), a straightforward implementation of this approach has a large performance penalty (see [FV1] for a detailed analysis). Significant improvements over the naive method have been recently obtained. In particular, a promissing approach consists in combining multiprecision integer (or rational) arithmetic and a floating-point filter based on interval analysis [KLN], [FV1], [BJMM].

Our approach falls in the exact integer arithmetic category and our objective is to use as few bits as possible to evaluate signs of determinants. Typically, we use no more bits than the number b of bits used to code the entries. This implies that we would not do multiplications of the entries and, *a fortiori*, we would not compute any determinants when we evaluate the signs.

To the best of our knowledge, the only related attempt has been made by Clarkson [Cla]. Clarkson uses an adaptation of the Gram–Schmidt procedure for computing an orthogonal basis, and employs approximate arithmetic. For a $d \times d$ determinant with b-bit integer entries, Clarkson's algorithm runs in time $O(d^3 b)$ and uses $2b + 1.5d$ bits to represent the values. Our algorithm is quite different. It is limited to 2×2 and 3×3 determinants and its asymptotic worst-case complexity is worse than that of Clarkson. However, it is simpler, it uses respectively b and $(b + 1)$-bit representations, and extensive simulations have shown that it performs well in practice. Since the most common applications are two- and three-dimensional, our method is competitive in this range of parameters.

The paper is organized as follows. In Section 2 we discuss the two-dimensional case in detail, both because it provides insights for the three-dimensional case and because it is used to resolve the determination of the sign of a three-dimensional determinant in Section 3. In Section 4 we present some significant applications of the outlined technique in computational geometry. In Section 5 we present and discuss experimental results and compare our algorithm with the straightforward computation using floating-point arithmetic or other exact arithmetics.

2. Two-Dimensional Case

2.1. *The Algorithm*. Let

$$D = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}$$

be a 2×2 determinant whose entries are b-bit integers.

If one of the four entries is zero, D reduces to the product of two integers and the sign of D follows from the sign of the two integers. In the rest of this section we assume that all four entries are nonzero.

Without loss of generality, we may assume that all the entries are strictly positive. Indeed, if an odd number of entries are negative, then the sign of D is trivially obtained. If two entries are negative, either they belong to the same row or the same column, in which case changing their sign changes the sign of D, or they belong to a diagonal, in which case changing their sign does not change D. If all entries are negative, changing their sign does not change D.

Furthermore, we may assume without loss of generality that $x_2 \geq x_1$ and $y_2 \geq y_1$. The case where $x_2 < x_1$ and $y_2 < y_1$ can be transformed to the previous one by exchanging the two rows of the matrix, which yields a change of the sign of D. In the other cases the sign of D can be obtained readily; specifically, if $x_1 \leq x_2$ and $y_2 < y_1$, $D < 0$, and, if $x_1 > x_2$ and $y_2 \geq y_1$, $D > 0$.

Under the above assumptions, we can write

$$x_2 = x_1 k_1 + x_r \qquad \text{with} \quad k_1 \in \mathbb{N} \quad \text{and} \quad 0 \leq x_r < x_1$$

and define

$$y_r = y_2 - k_1 y_1.$$

Then

$$D = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_2 - k_1 x_1 & y_2 - k_1 y_1 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_r & y_r \end{vmatrix}.$$

If $y_1 > 2^b/k_1$, then y_r cannot be computed, but in that case y_r is certainly negative and thus $D < 0$ otherwise y_r can be computed. If it is outside the range $[0, y_1]$ the sign of D can be obtained too: if $y_r < 0$, then $D < 0$ and if $y_r > y_1$, then $D > 0$. Moreover, if $x_r < x_1/2$ and $y_r > y_1/2$, then $D > 0$, and if $x_r > x_1/2$ and $y_r < y_1/2$, then $D < 0$. Otherwise we rewrite D as follows:

$$\text{if} \quad x_r < \frac{x_1}{2} \quad \text{and} \quad y_r < \frac{y_1}{2}, \quad \text{then} \quad D = \begin{vmatrix} x_1 & y_1 \\ x_r & y_r \end{vmatrix},$$
$$\text{if} \quad x_r > \frac{x_1}{2} \quad \text{and} \quad y_r > \frac{y_1}{2}, \quad \text{then} \quad D = \begin{vmatrix} x_1 & y_1 \\ x_1 - x_r & y_1 - y_r \end{vmatrix}.$$

In both cases we get a new determinant where both entries of the second row have been divided by at least two, and we can iterate the computation.

In conclusion, at each iteration, using only comparisons and euclidean divisions, either the algorithm stops or it iterates on a reduced problem, where a row is replaced by one whose entries are less than half the size of the original ones. Hence, the number of iterations is bounded from above by the logarithm of the largest representable integer, i.e., the number b of bits in the binary representation of the initial entries.

We sum up the results in the following theorem.

THEOREM 1. *Let D be a 2×2 determinant with b-bit integer entries. There exists an algorithm that evaluates the sign of D using only b-bit arithmetic. The algorithm requires at most b iterations, each iteration involving $O(1)$ additions/subtractions, comparisons and euclidean divisions.*

3. Three Dimensions

3.1. *Geometric Intuition*. Let

$$D = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

be a 3×3 determinant with b-bit integer entries. Vector (x_i, y_i, z_i) is denoted U_i and the unit vectors along the three axis are denoted E_x, E_y, and E_z. The z direction is called *vertical* and u denotes the vertical projection of vector U onto the horizontal plane $z = 0$. For convenience, we often identify a vector U with the point whose coordinate vector is U. Without loss of generality, we can assume that the z_i are nonnegative and that $z_3 \geq z_1, z_2$.

The basic idea is as follows: unless the sign of D can be directly assessed, we replace the original matrix with a matrix having the *same determinant*, but provably smaller entries. This assures that the evaluation will terminate. The basic device used is the standard addition to a row of a linear combination of the other rows.

More specifically, assume (here and in Sections 3.1–3.6), that the projections u_1, u_2 of U_1, U_2 are noncolinear (i.e., independent) vectors. This implies that the three vectors U_1, U_2, and E_z are linearly independent, and we can express U_3 as

$$U_3 = \kappa_1 U_1 + \kappa_2 U_2 + \kappa_3 E_z, \tag{1}$$

where $\kappa_1, \kappa_2, \kappa_3 \in \mathbb{R}$. It follows that

$$D = \kappa_3 \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ 0 & 0 & 1 \end{vmatrix}. \tag{2}$$

Hence, if the sign of κ_3 is known, the problem is reduced to evaluating the sign of

$$\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix},$$

and we are faced with a two-dimensional problem.

We now show that, in some (usually most) cases, the sign of κ_3 can be determined. We have

$$\kappa_3 = z_3 - \kappa_1 z_1 - \kappa_2 z_2. \tag{3}$$

Let $\kappa_1 = k_1 + \rho_1$, $\kappa_2 = k_2 + \rho_2$, with $k_1 = \lfloor \kappa_1 \rfloor$, $k_2 = \lfloor \kappa_2 \rfloor$, and $0 \leq \rho_1, \rho_2 < 1$. We define

$$R = U_3 - k_1 U_1 - k_2 U_2.$$

Then (3) becomes

$$\kappa_3 = z_R - \rho_1 z_1 - \rho_2 z_2.$$

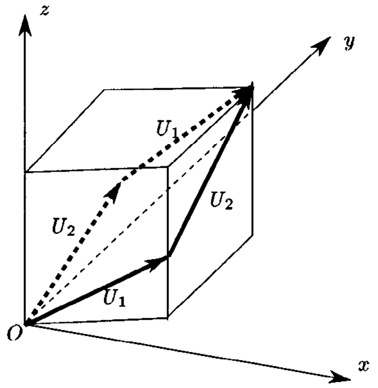

Fig. 1. Box $\mathcal{B}$.

If $z_R < 0$, then $\kappa_3 < 0$ since z_1 and z_2 are nonnegative, and, if $z_R > z_1 + z_2$, $\kappa_3 > 0$. Otherwise, R lies in the intersection $\mathcal{B}$ of the cylinder projecting along the z axis onto the parallelogram $u_1 \oplus u_2$ ($\oplus$ denoting the Minkowski sum) with the slab of points W such that $0 \le z_W \le z_1 + z_2$ (see Figure 1). In this case, we still do not know the sign of κ_3, but we can write:

(4)
$$D = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_R & y_R & z_R \end{vmatrix},$$

where the row with the largest z component has been replaced by R.

The following geometric interpretation of our definitions will be useful in what follows. Let H be the plane passing through O and spanned by U_1 and U_2. The vectors U_1 and U_2 generate in H the lattice $\mathcal{L}_H = \{l_1U_1 + l_2U_2, l_1, l_2 \in \mathbb{Z}\}$. $\mathcal{L}_H$ projects vertically onto the lattice $\mathcal{L}$ of the horizontal plane generated by u_1 and u_2. To each cell $\mathcal{C} = \{(l_1 + \varepsilon_1)u_1 + (l_2 + \varepsilon_2)u_2,\ 0 \le \varepsilon_1, \varepsilon_2 \le 1\}$ of $\mathcal{L}$, we associate its *reference point* $l_1u_1 + l_2u_2$. In particular, $k_1u_1 + k_2u_2$ is the reference point of the cell of $\mathcal{L}$ that contains u_3. To each cell of $\mathcal{L}$, we associate also a *box*. The box associated to the cell $\mathcal{C}$ of $\mathcal{L}$ with reference point $l_1u_1 + l_2u_2$ is a copy of $\mathcal{B}$ translated by vector $l_1U_1 + l_2U_2$ which projects vertically onto $\mathcal{C}$. Let $\mathbb{B}$ be the union of these boxes, i.e., the translated copies of $\mathcal{B}$ by the vectors $l_1U_1 + l_2U_2$ for $l_1, l_2 \in \mathbb{Z}$. $\mathbb{B}$ contains plane H and can be considered as an approximation of H.

The geometric interpretation of the above dicussion is now: if U_3 lies above (resp. below) $\mathbb{B}$, κ_3 is positive (resp. negative), and otherwise, U_3 can be replaced by a vector contained in $\mathcal{B}$.

3.2. *The Algorithm.* The algorithm consists of $O(b)$ iterations, but may terminate earlier. Each iteration consists of three steps.

During a *preliminary step*, described in Section 3.3, the vectors whose z components are negative are replaced by the opposite vectors and some easy cases are solved.

The *first step*, described in Section 3.4, determines whether U_3 lies below $\mathbb{B}$, above $\mathbb{B}$, or inside some box of $\mathbb{B}$. In the first two cases D reduces to a 2×2 determinant with b-bit integer entries: its sign can be evaluated using the algorithm of Section 2. In the last case we translate point U_3 in a direction parallel to H to obtain $R = U_3 - k_1U_1 - k_2U_2 \in \mathcal{B}$. However, we cannot simply iterate on the vectors (U_1, U_2, R). Indeed, although R is known to lie in box $\mathcal{B}$, the binary representation of its components may require as many as $b + 1$ bits. Furthermore, the z component z_R of R only satisfies $0 \le z_R \le z_1 + z_2$, which does not imply that this component is smaller than the original z_3.

The *second step* of the algorithm, to be described in Section 3.6, will either evaluate the sign of κ_3 or find a vector $R' = R + \theta_1U_1 + \theta_2U_2$ $(\theta_1, \theta_2 \in \{-1, 0, +1\})$ such that the encoding length of its x and y components does not exceed b, and its z component is less than $z_3/2$.

The algorithm is then iterated on the vectors (U_1, U_2, R'). Now, a reduction by two of the modulus of the maximum z component of the vectors in D is guaranteed after at most three iterations. Hence, in total, at most $3b$ iterations will be required either to find that $D = 0$ or end up with a 2×2 determinant.

3.3. *Preliminary Step.* As in the two-dimensional case, the sign of the determinant can be evaluated readily in some cases. First, we multiply by -1 the rows whose z component are negative so that all the entries of the last column are nonnegative. This does not change D if no or two rows are concerned and changes D to $-D$ otherwise. Secondly, we permute the rows so that U_3 has the largest z component. Again the change in the sign of D induced by the permutation is known.

Then, if the three minors

$$\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}, \begin{vmatrix} x_2 & y_2 \\ x_3 & y_3 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} x_3 & y_3 \\ x_1 & y_1 \end{vmatrix}$$

are all strictly positive (resp. negative), then D is positive (resp. negative). Geometrically, this case corresponds to the situation where u_1, u_2, and u_3 span positively the horizontal plane, or, equivalently, the origin lies inside the triangle $u_1u_2u_3$.

3.4. *First Step*: *Computing R*. The first step of each iteration either determines the sign of κ_3 or, when U_3 lies in $\mathbb{B}$, computes the vector $R = U_3 - k_1U_1 - k_2U_2$. This is not an obvious task since the moduli of the integers k_1 and k_2 can be as large as 2^{2b+1}: indeed, $k_1 = \lfloor \kappa_1 \rfloor$ and $k_2 = \lfloor \kappa_2 \rfloor$ and it follows from (1) that

$$\kappa_1 = \frac{\begin{vmatrix} x_3 & y_3 \\ x_2 & y_2 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}} \quad \text{and} \quad \kappa_2 = \frac{\begin{vmatrix} x_1 & y_1 \\ x_3 & y_3 \end{vmatrix}}{\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}}.$$

As we restrict ourselves to $(b + 1)$-bit arithmetic (a full discussion of that point is given in Section 3.5), we may not be able to compute the k_i.

Consider again the lattice $\mathcal{L}$ in the horizontal plane generated by u_1 and u_2. As vector U_3 has b-bit integer components, the coordinates of the points of the line segments OU_3 and Ou_3 have moduli less than 2^b.

Informally, our strategy to compute R is to probe a subset of $O(b)$ edges of the lattice $\mathcal{L}$ crossed by Ou_3. For each such edge, we consider the corresponding edge in lattice

$\mathcal{L}_H$. If the z range of all edges encountered during the process remains inside the bounds of the available $(b+1)$-bit arithmetic, the procedure ends when the cell containing u_3 is found. If, on the contrary, an encountered edge of $\mathcal{L}_H$ belongs to a box whose z range exceeds the $(b+1)$-bit representation, the z range of this edge extends entirely outside the bounds of the b-bit representation. In such a case the sign of κ_3 can be determined easily and the calculation of R is halted.

We now present the details of the procedure.

Substep 1.1. Consider the parallelogram formed by the union of the four lattice cells $\mathcal{C}_0$, $\mathcal{C}_0 - u_1, \mathcal{C}_0 - u_2, \mathcal{C}_0 - (u_1 + u_2)$, where $\mathcal{C}_0$ is the Minkowski sum $u_1 \oplus u_2$. The boundary of this parallelogram consists of eight cell edges. By a straightforward (three-step) binary search we determine which of these edges is intersected by the half-line L issued from O and containing u_3, the search discriminant being the sign of a 2×2 determinant of the form $\begin{vmatrix} u_3 \\ w \end{vmatrix}$ where $w \in \{u_1, u_2, u_1 + u_2, u_1 - u_2\}$. This search also identifies which of the cells $\mathcal{C}_0$, $\mathcal{C}_0 - u_1$, $\mathcal{C}_0 - u_2$, $\mathcal{C}_0 - (u_1 + u_2)$ is intersected by the half-line L. Let $\mathcal{C}_0' = \mathcal{C}_0 - \varepsilon_1 u_1 - \varepsilon_2 u_2$ be such cell. Note that if any of the determinants $\begin{vmatrix} u_3 \\ w \end{vmatrix}$ is equal to zero, then either $k_1 = \pm k_2$ or one of the k_i is zero. In such cases the sign of κ_3 or R can be computed as in the two-dimensional case.

For the sake of simplicity, we standardize the problem by replacing the original basis (u_1, u_2) with the basis (v_1, v_2), where $v_i = (-2\varepsilon_i + 1)u_i$, $i = 1, 2$. Denoting by c_u (resp. c_v) the reference point of the cell $\mathcal{C}$ of $\mathcal{L}$ that contains u_3 when we take (u_1, u_2) (resp. (v_1, v_2)) as basis vectors of $\mathcal{L}$, we observe that

$$c_u = c_v + \varepsilon_1 u_1 + \varepsilon_2 u_2. \tag{5}$$

We in fact compute c_v, and then obtain c_u using (5). In what follows k_1' and k_2' denote the coordinates of c_v in the basis (v_1, v_2). Let D_i, $i = 1, 2$, be the line of the horizontal plane containing v_i.

Substep 1.2. Next, we have to test whether or not u_3 belongs to cell $\mathcal{C}_0'$. This test entails locating u_3 with respect to the edge traversed by the half-line L, i.e., locating u_3 with respect to either $D_1 + v_2$ or $D_2 + v_1$. In the first case we evaluate the sign of $\begin{vmatrix} v_1 \\ u_3 - v_2 \end{vmatrix}$ and, in the second, of $\begin{vmatrix} u_3 - v_1 \\ v_2 \end{vmatrix}$ (if the sign is negative, $u_3 \in \mathcal{C}_0'$). If u_3 is included in $\mathcal{C}_0'$, $k_1' = k_2' = 0$, we deduce c_u and the corresponding point C_u of $\mathcal{L}_H$ using (5), compute $R = U_3 - C$ and go to Substep 1.6. Otherwise, we proceed to Substep 1.3.

Substep 1.3. Let $K(1)$ denote the first lattice line traversed by Ou_3, i.e., $K(1) \in \{D_1 + v_2, D_2 + v_1\}$. Without loss of generality, here and hereafter, we assume $K(1) = D_2 + v_1$ (see Figure 2). For $\lambda \in \mathbb{N}$, we note $K(\lambda)$ the line $D_2 + \lambda v_1$. The vertices of the edge of lattice $\mathcal{L}$ belonging to $K(\lambda)$ and intersected by Ou_3 (if such an intersection exists) are denoted $c(\lambda)$ and $c(\lambda) + v_2$. Notice that $c(\lambda) = \lambda v_1 + \lambda' v_2$ for some $\lambda' \in \mathbb{N}$, $0 \leq \lambda' < \lambda$.

Notice that like k_1 and k_2, λ and λ' may be as large as 2^{2b+1} but since they would not be explicitly computed this is not a problem.

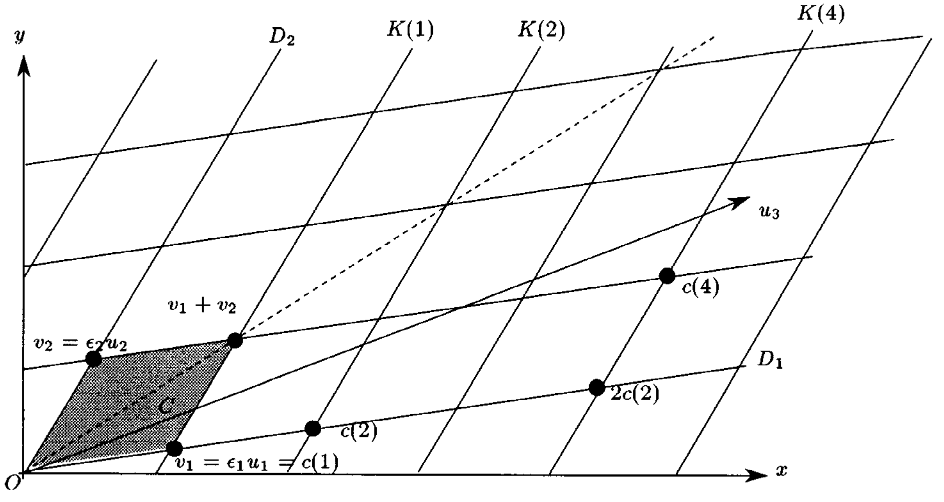

Fig. 2. Illustration of Step 1.

In this substep we successively probes lines $K(2)$, $K(4), \ldots, K(2^i), \ldots$. Each of these probes may yield the sign of κ_3, in which case the process terminates. If not, the search continues until the unique integer k is determined such that Ou_3 intersects $K(2^k)$ but not $K(2^{k+1})$. After completing the probe of $K(2^j)$, the algorithm stores in a stack S point $c(2^j)$. It is convenient to describe the search as a sequence of simpler actions detailed below.

1.3.1. Assume that $c(2^{l-1})$ has been determined. Ou_3 crosses the line $K(2^{l-1})$ between $c(2^{l-1})$ and $c(2^{l-1})+v_2$. Let z' and z'' be the z coordinates of the two corresponding points of the lattice $\mathcal{L}_H$, $|z'-z''|=z_2$. If z' and z'' are both negative, then $\kappa_3>0$ and, if z' and z'' are both greater than 2^b, then $\kappa_3<0$; in such cases the sign of D is known and the algorithm halts. Otherwise z' and z'' have both encoding lengths at most $b+1$, and the process continues.

1.3.2. We test if Ou_3 crosses the line $K(2^l)$ by evaluating the sign of the 2×2 determinant

$$\Delta = \begin{vmatrix} u_3 - 2c(2^{l-1}) \\ v_2 \end{vmatrix}.$$

Notice that $2c(2^{l-1}) \in K(2^l)$. If Ou_3 does not intersect $K(2^l)$ ($\Delta<0$), then we go to Substep 1.4. Otherwise, we compute $c(2^l)$ as described in Substep 1.3.3.

1.3.3. Since Ou_3 crosses $K(2^{l-1})$ between $c(2^{l-1})$ and $c(2^{l-1})+v_2$, Ou_3 crosses $K(2^l)$ between $2c(2^{l-1})$ and $2c(2^{l-1})+2v_2$. Clearly, $c(2^l)$ is either $2c(2^{l-1})$ or $2c(2^{l-1})+v_2$ (the latter in Figure 3). Point $c(2^l)$ can be determined by testing on which side of Ou_3 point $2c(2^{l-1})+v_2$ lies, i.e., by evaluating the sign of the 2×2 determinant $\begin{vmatrix} u_3 \\ 2c(2^{l-1})+v_2 \end{vmatrix}$. Go to 1.3.1.

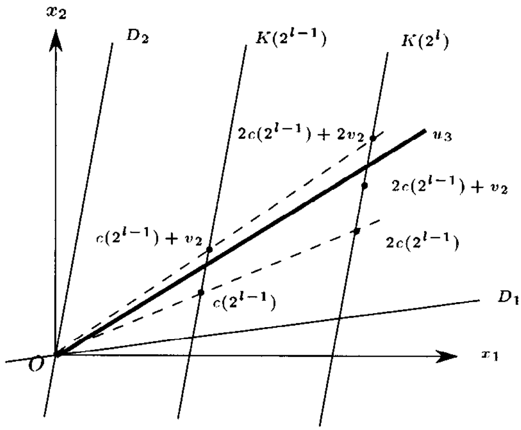

Fig. 3. For Substep 1.3.3.

Substep 1.4. Assume now that Ou_3 intersects the line $K(2^k)$ but not the line $K(2^{k+1})$. Then the algorithm computes the integer λ_f such that Ou_3 crosses $K(\lambda_f)$ but not $K(\lambda_f + 1)$. The determination of λ_f is a binary search. This search involves k steps numbered $k-1, k-2, \ldots, 0$. We denote λ_h the integer such that Ou_3 intersects $K(\lambda_h)$ but not $K(\lambda_h + 2^h)$. Assume that at the beginning of step h, we know λ_{h+1} and $c(\lambda_{h+1})$. We now explain how λ_h and $c(\lambda_h)$ are inductively computed from λ_{h+1} and $c(\lambda_{h+1})$. The basis of the induction is given by $h + 1 = k$ and $\lambda_{h+1} = 2^k$. Again, the search is better illustrated as a sequence of simpler actions.

1.4.1 First, as in Step 1.3.1, let z' and z'' be the z coordinates of the two points of the lattice $\mathcal{L}_H$ corresponding to $c(\lambda_{h+1})$ and $c(\lambda_{h+1}) + v_2$. If z' and z'' are both negative, then $\kappa_3 > 0$ and, if z' and z'' are both greater than 2^b, then $\kappa_3 < 0$; in such cases, the sign of D is known and the algorithm halts. Otherwise z' and z'' have both encoding lengths at most $b + 1$.

1.4.2 The algorithm then determines on which side of $K(\lambda_{h+1} + 2^h)$ point u_3 lies, which can be done by evaluating the sign of the 2×2 determinant

$$\Delta' = \begin{vmatrix} u_3 - c(\lambda_{h+1}) - c(2^h) \\ v_2 \end{vmatrix},$$

where $c(2^h)$ is popped out of stack S. If u_3 lies on the same side of $K(\lambda_{h+1} + 2^h)$ as O ($\Delta' < 0$), then $\lambda_{h+1} = \lambda_h$, $c(\lambda_h) = c(\lambda_{h+1})$, and we can proceed to Step $h - 1$. Otherwise, $\lambda_h = \lambda_{h+1} + 2^h$ and we have to compute $c(\lambda_h)$.

1.4.3 It is to be observed that Ou_3 crosses $K(\lambda_h)$ between $c(\lambda_{h+1}) + c(2^h)$ and $c(\lambda_{h+1}) + c(2^h) + 2v_2$. Thus $c(\lambda_h)$ is either $c(\lambda_{h+1}) + c(2^h)$ or $c(\lambda_{h+1}) + c(2^h) + v_2$ depending on which side of Ou_3 point $c(\lambda_{h+1}) + c(2^h) + v_2$ lies, which is given by the sign of $\begin{vmatrix} u_3 \\ c(\lambda_{h+1}) + c(2^h) + v_2 \end{vmatrix}$. Go to 1.4.1.

The search is carried on until one of the following occurs: the algorithm halts, or it goes to Substep 1.5 at an intermediate stage, or it performs all the steps $k-1, k-2, \ldots, 0$. In the last case, we set $\lambda_f = \lambda_0$ and go to Substep 1.5.

Substep 1.5. This substep determines the cell of $\mathcal{L}$ that contains u_3. We know that u_3 belongs to the parallelogram $c(\lambda_f), c(\lambda_f)+v_1, c(\lambda_f)+v_1+2v_2, c(\lambda_f)+2v_2$. A last test locates u_3 with respect to line $D_1 + c(\lambda_f) + v_2$ and determines the cell $\mathcal{C}$ of the lattice $\mathcal{L}$ that contains u_3. More precisely, if

$$\begin{vmatrix} u_3 - c(\lambda_f) - v_2 \\ v_1 \end{vmatrix} \geq 0,$$

then $\mathcal{C}$ is the cell whose reference point is $c(\lambda_f)$ in the (v_1, v_2) basis. Otherwise, $\mathcal{C}$ is the cell whose reference point is $c(\lambda_f) + v_2$. The reference point c_u of $\mathcal{C}$ in the (u_1, u_2) basis and the corresponding point C_u of $\mathcal{L}_H$ are then obtained using (5) and we compute $R = U_3 - C_u$.

Substep 1.6. If $z_R < 0$, then $\kappa_3 < 0$ and, if $z_R > z_1 + z_2$, then $\kappa_3 > 0$. Otherwise, we go to Step 2 (described in Section 3.6).

This ends the description of the first major step of the algorithm. The vector $r = u_3 - c_u$ belongs to the cell of $\mathcal{L}$ whose reference point is O, thus the encoding lengths of x_R and y_R are at most $b+1$ and $0 \leq z_R \leq z_1 + z_2$.

3.5. *Arithmetic*. In this section we show that $(b+1)$-bit arithmetic is sufficient to run the above algorithm, assuming that U_1, U_2, and U_3 are b-bit integers.

First, we observe that all encountered $c(\lambda)$ and $c(\lambda) + v_2$ can be represented using $b+1$ bits. Indeed, since $p = K(\lambda) \cap Ou_3$ belongs to the line segment Ou_3, the encoding length of its coordinates is less than the one of u_3, and since $c(\lambda)$ and $c(\lambda)+v_2$ belong to the line segment $p-u_2$, $p+u_2$, one additional bit is enough to store $c(\lambda)$ and $c(\lambda)+v_2$. In the case where $c(\lambda)$ is computed as the sum of two terms $c(\lambda_1)+c(\lambda_2)$, then clearly $c(\lambda)$ can be computed without difficulty. In the other case where $c(\lambda) = [c(\lambda_1)+c(\lambda_2)]+v_2$, $(b+2)$ bits may be required to store the intermediate result $c(\lambda_1)+c(\lambda_2)$, but, fortunately, among the three possibilities $c(\lambda) = [c(\lambda_1)+c(\lambda_2)]+v_2$, $c(\lambda) = c(\lambda_1)+[c(\lambda_2)+v_2]$, and $c(\lambda) = c(\lambda_2) + [c(\lambda_1) + v_2]$ there always exists one that allows us to compute the x (resp. y, z) coordinate of the sum so that the intermediate result remains on $b+1$ bits (the formulas may be different for the different coordinates).

Next, we show that the computations of vector R performed in Substeps 1.2 or 1.5 do not require more than $(b+1)$-bit arithmetic. The computation of the x- and y-components of R do not cause any problem since $c_v - c_u$, $u_3 - c_v$, and $u_3 - c_u$ can each be represented with at most $b+1$ bits. Consider now the computation of z_R. Let z_v be the z-component of the point C_v of $\mathcal{L}_H$ which corresponds to c_v and let z_u be the z-component of C_u. If $z_R = z_3 - z_u = z_3 - z_v - \varepsilon_1 z_1 - \varepsilon_2 z_2$ is in the range $[-2^{b+1}, 2^{b+1}]$; then can be computed (indeed there is a way to organize the above sum in order to compute it using $(b+1)$-bit arithmetic); else $(|z_R| > 2^{b+1})$, we can easily decide if z_R is positive or negative (κ_3 has the same sign as z_R).

The bit-length of z_v is known to be at most $b+1$, and it is easy to see from (5) that the bit-length of z_u does not exceed the bit-length of z_v. If z_u is positive or negative with $|z_u| < 2^b$, $z_R = z_3 - z_u$ can be computed using $(b+1)$-bit arithmetic. If z_u is negative with $|z_u| > 2^b$, then z_R and κ_3 are known to be negative and we do not need to compute z_R.

At last we show that the signs of the determinants used in Steps 1.3.2, 1.3.3, 1.4.2, and 1.4.3 can be evaluated using $(b+1)$-bit arithmetic. The vectors appearing in those determinants are combinations of u_1, u_2, u_3, and some $c(\lambda)$: for instance, at Step 1.3.3, we need to compute $2c(2^{l-1}) + v_2$. It follows from the description of the algorithm that these vectors can always be computed using $(b+3)$-bit arithmetic. We now explain how to reduce the number of bits of the arithmetic to $b+1$ for each of the Steps 1.3.2, 1.3.3, 1.4.2, and 1.4.3 of the algorithm.

Step 1.3.2. In this step we test on which side of $K(2^l)$ point u_3 lies. The problem arises when the coordinates of $u_3 - 2c(2^{l-1})$ are not both representable with $b+1$ bits. The idea is therefore to replace point $2c(2^{l-1})$ with another point w of $K(2^l)$ such that $u_3 - w$ is represented with $b+1$ bits. Let $\mathcal{SP} \triangleq [-2^b, 2^b] \times [-2^b, 2^b]$ be the single precision domain. We claim that such a construction can be accomplished if at least one of $\{c(2^{l-1}), c(2^{l-1}) + v_2\}$ and at least one of $\{c(2^{l-1}) - u_3, c(2^{l-1}) + v_2 - u_3\}$ belong to $\mathcal{SP}$. Let w_1 and $w_2 - u_3$ be points in $\mathcal{SP}$ chosen from these two sets, respectively. It follows that point $-w_1 - (w_2 - u_3) = u_3 - (w_1 + w_2)$ is representable with $b+1$ bits and that $w_1 + w_2 \in K(2^l)$ since $w_1, w_2 \in K(2^{l-1})$. We conclude that the entries of the determinant $\begin{vmatrix} u_3 - w \\ v_2 \end{vmatrix}$ are representable with $b+1$ bits. Otherwise we have the following easily decidable alternatives:

(i) $c(2^{l-1}) \notin \mathcal{SP}$ and $c(2^{l-1}) + v_2 \notin \mathcal{SP}$. We claim that u_3 and O are on the same side of $K(2^l)$. Indeed, since $v_2 \in \mathcal{SP}$, the line $K(2^{l-1})$ does not intersect the square $\mathcal{SP}^*$ with vertices $(0, \pm 2^b)$ and $(\pm 2^b, 0)$ (see Figure 4(a)). Then $K(2^l)$ is entirely outside the square $2\mathcal{SP}^*$ which contains $\mathcal{SP}$.

(ii) $c(2^{l-1}) - u_3 \notin \mathcal{SP}$ and $c(2^{l-1}) + v_2 - u_3 \notin \mathcal{SP}$. We claim that u_3 and O are on distincts sides of $K(2^l)$. Indeed, arguing as above, line $K(2^{l-1})$ does not intersect the square $\mathcal{SP}^* + u_3$ ($\mathcal{SP}^*$ centered at u_3, see Figure 4(b)). Thus, if p is the intersection between $K(2^{l-1})$ and Ou_3, $||pu_3||_\infty \geq \frac{1}{2}||pu_3||_1 > 2^{b-1}$ (since p, being external to $\mathcal{SP}^* + u_3$, has L_1-distance from $u_3 > 2^b$). The point $p' = 2p$ is the point where the lines Ou_3 and $K(2^l)$ intersect and we have

$$\begin{aligned} ||Op'||_\infty &= 2\,||Op||_\infty = 2\,||Ou_3||_\infty - 2\,||pu_3||_\infty \\ &< 2\,||Ou_3||_\infty - 2^b \leq ||Ou_3||_\infty\,, \end{aligned}$$

which proves the claim.

Step 1.4.2. The modification of this step is similar to (and simpler than) that of Step 1.3.2. Here we are led to the evaluation of the sign of $\begin{vmatrix} u_3 - w \\ v_2 \end{vmatrix}$, where point w must be chosen on line $K(\lambda_{h+1} + 2^h)$. Since, as we already know, u_3 and O are on distinct sides of $K(2^{h+1})$, one of $c(2^h)$ and $c(2^h) + v_2$ can be represented with b bits; let w_1 be this vector. Similary, since O and u_3 are on the same side of $K(2\lambda_{h+1})$, one of $c(\lambda_{h+1}) - u_3$ and $c(\lambda_{h+1}) + v_2 - u_3$ can be represented with b bits; let $w_2 - u_3$ be this

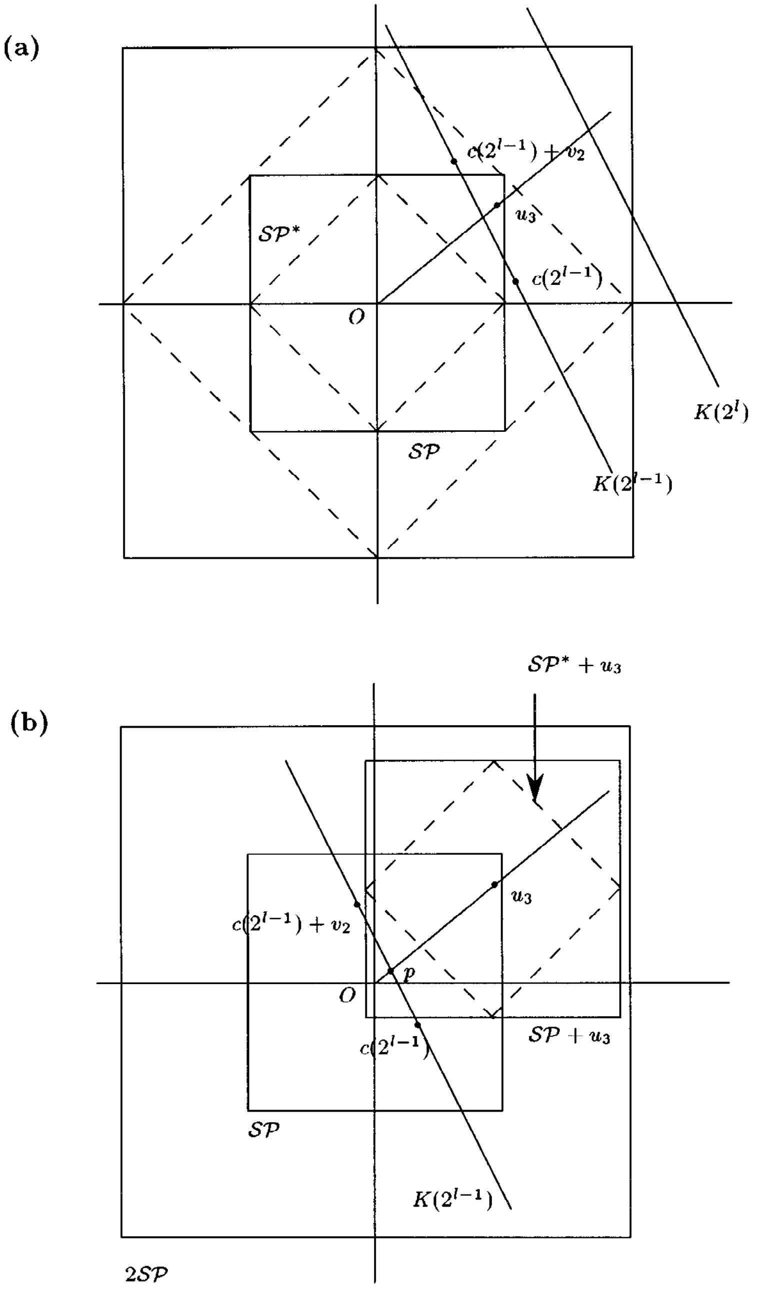

Fig. 4. Illustrations for the cases: (a) $c(2^{l-1}) \notin \mathcal{SP}$ and $c(2^{l-1}) + v_2 \notin \mathcal{SP}$, and (b) $c(2^{l-1}) - u_3 \notin \mathcal{SP}$ and $c(2^{l-1}) + v_2 - u_3 \notin \mathcal{SP}$.

vector. The opposite of the sum of these two vectors $-(w_1+(w_2-u_3)) = u_3-(w_1+w_2)$ is represented with $b+1$ bits and $w \triangleq w_1 + w_2 \in K(\lambda_{h+1} + 2^h)$, as we wished to show.

Steps 1.3.3 *and* 1.4.3. The entries of the determinant $\begin{vmatrix} u_3 \\ 2c(2^{l-1}) + v_2 \end{vmatrix}$ used in Step 1.3.3 can be represented with $b+1$ bits since $2c(2^{l-1}) + v_2$ is either $c(2^l)$ or $c(2^l) + v_2$. The same holds for the entries of the determinant used at Step 1.4.3.

3.6. *Second Step*: *Exponential Reduction*. The second major step of the algorithm, to be described in this subsection, either evaluates the sign of κ_3 or finds a vector $R' = R-\theta_1 U_1 - \theta_2 U_2, \theta_1, \theta_2 \in \{-1, 0, 1\}$ such that $x_{R'}, y_{R'}$ are b-bit integers and $|z_{R'}| \le z_3/2$. In order to do that, we further subdivide the box $\mathcal{B}$ into four subboxes such that, for any r lying in each of those subboxes, either such a vector R' can be determined or the sign of κ_3 can be readily obtained.

The projection of box $\mathcal{B}$ onto the plane is the parallelogram $u_1 \oplus u_2$. The algorithm locates the projection r of R with respect to the two diagonals of the parallelogram joining u_1 to u_2 and O to $u_1 + u_2$ by evaluating the sign of 2×2 determinants

$$\begin{vmatrix} r \\ u_1 + u_2 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} r - u_1 \\ u_2 - u_1 \end{vmatrix}$$

(see Figure 5).

We illustrate the step just for one of the four subboxes; handling of the other three cases is trivially analogous. Assume therefore, without loss of generality, that r belongs to triangle $(O, u_1, (u_1 + u_2)/2)$ (the shaded triangle in Figure 5). Then

- if $z_R > \sup(z_1, (z_1 + z_2)/2)$, then $\kappa_3 > 0$,
- else, we choose R' as the vector among the two vectors R and $R - U_1$ whose z

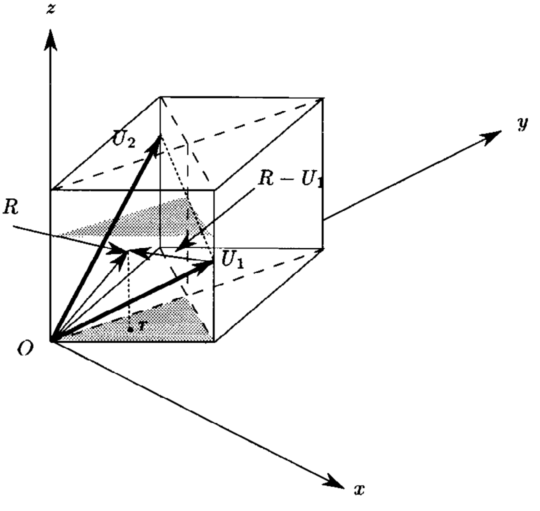

Fig. 5. Illustration of the selection of R'.

component has the smaller modulus. It follows that

$$|z_{R'}| \le \frac{\sup(z_1, z_2)}{2} \le \frac{z_3}{2}.$$

3.7. *Nonindependent Vectors.* We have assumed above that (u_1, u_2) are linearly independent vectors. If these vectors are not independent, the minor $\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}$ of

$$D = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}$$

associated to the z_3 component of u_3 vanishes and we can simply replace z_3 by zero without modifying the determinant:

$$D = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & 0 \end{vmatrix}.$$

3.8. *Complexity Analysis of the Algorithm.* At each iteration, either the algorithm evaluates the sign of κ_3 and ends by the computation of a 2×2 determinant, or it iterates with a 3×3 determinant where the greatest (in absolute value) element of the last column has been divided by at least two while the others remain unchanged. It follows that the number of iterations is at most $3b$.

Each iteration consists of Substeps 1.1–1.6 and Step 2. The cost of each step is dominated by evaluating signs of 2×2 determinants with $(b + 1)$-bit integer entries. Substep 1.1 requires three such evaluations, Substep 1.2 one, Substep 1.3 at most $4b$ (each iteration requires evaluating the sign of two determinants and there are at most $\log k_i \le 2b$ iterations), Substep 1.4 at most $4b$, Substep 1.5 at most one, and Substep 2 at most two. Thanks to Theorem 1, we have the following theorem.

THEOREM 2. *Let D be a 3×3 determinant with b-bit integer entries. There exists an algorithm that evaluates the sign of D using only $(b + 1)$-bit arithmetic. The algorithm requires at most $3b$ iterations, each iteration involving the evaluation of at most $8b + 9$ signs of 2×2 determinants with $(b+1)$-bit integer entries. In the worst case the algorithm requires at most $3b^2(8b + 9)$ elementary steps, each elementary step involving $O(1)$ additions/subtractions, comparisons, and euclidean divisions.*

4. Geometric Applications. Most geometric tests can be reduced to computing the sign of a determinant. This section shows such reductions for the most basic geometric tests.

4.1. `which_side`. Given d points $A_1, \ldots, A_d$ in d-space, and another point X of $\mathbb{R}^d$, Function `which_side` determines whether X belongs to the hyperplane H passing through the A_i or, otherwise, to which half-space limited by H. This function is at the core of many geometric algorithms, and, in particular, of all algorithms computing convex hulls of points in $\mathbb{R}^d$.

Function `which_side` can be implemented as the evaluation of the sign of the $d \times d$ determinant

$$D = \begin{vmatrix} A_1 - X \\ \vdots \\ A_d - X \end{vmatrix}$$

with $(b+1)$-bit integer entries, if the points have b-bit integer coordinates.

4.2. `sign_dot_product`. Given two 2-vectors $U_1 = (x_1, y_1)$ and $U_2 = (x_2, y_2)$ with b-bit integer components, the sign of the dot product $U_1 \cdot U_2$ can be determined by determining the sign of the 2×2 determinant with b-bit integer entries

$$\begin{vmatrix} y_1 & -x_1 \\ x_2 & y_2 \end{vmatrix} = U_1 \cdot U_2.$$

Given two 3-vectors $U_1 = (x_1, y_1, z_1)$ and $U_2 = (x_2, y_2, z_2)$ with b-bit integer components, if $x_1 \neq 0$, the sign of the dot product $U_1 \cdot U_2$ can be determined by determining the sign of the 3×3 determinant with b-bit integer entries

$$\begin{vmatrix} y_1 & -x_1 & 0 \\ z_1 & 0 & -x_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = x_1\, U_1 \cdot U_2.$$

If $x_1 = 0$, the problem is reduced to the two-dimensional case (this formula can be generalized to higher dimensions).

These results immediately apply for comparing the norms of two vectors. Indeed, given two vectors U_1 and U_2,

$$|U_1|^2 - |U_2|^2 = (U_1 + U_2) \cdot (U_1 - U_2).$$

It follows that comparing the norms of two d-vectors with b-bit integer components, is equivalent to evaluating the sign of a $d \times d$ determinant with $(b+1)$-bit integer entries.

4.3. `in_circle`. The basic numerical test involved in the construction of Voronoi diagrams in the plane is the following. Given are three points $A_i = (x_i, y_i)$, $i = 1, 2, 3$, and another point $X = (x, y)$ with b-bit integer coordinates. The test consists in deciding whether X lies on the circle C passing through A_1, A_2, and A_3, inside C, or outside C. This is equivalent to determining the sign of the following 3×3 determinant:

$$\texttt{in\_circle}(X) = \begin{vmatrix} x_1 - x & x_2 - x & x_3 - x \\ y_1 - y & y_2 - y & y_3 - y \\ x_1^2 + y_1^2 - x^2 - y^2 & x_2^2 + y_2^2 - x^2 - y^2 & x_3^2 + y_3^2 - x^2 - y^2 \end{vmatrix}$$

$$= \begin{vmatrix} x_1 - x & x_2 - x & x_3 - x \\ y_1 - y & y_2 - y & y_3 - y \\ 0 & (x_2 - x)(x_2 - x_1) + (y_2 - y)(y_2 - y_1) & (x_3 - x)(x_3 - x_1) + (y_3 - y)(y_3 - y_1) \end{vmatrix}$$

$$= \frac{1}{x_1 - x} \begin{vmatrix} x_1 - x & x_2 - x & x_3 - x \\ 0 & (x_1 - x)(y_2 - y) - (x_2 - x)(y_1 - y) & (x_1 - x)(y_3 - y) - (x_3 - x)(y_1 - y) \\ 0 & (x_2 - x)(x_2 - x_1) + (y_2 - y)(y_2 - y_1) & (x_3 - x)(x_3 - x_1) + (y_3 - y)(y_3 - y_1) \end{vmatrix}$$

$$= \begin{vmatrix} (x_1 - x)(y_2 - y) - (x_2 - x)(y_1 - y) & (x_1 - x)(y_3 - y) - (x_3 - x)(y_1 - y) \\ (x_2 - x)(x_2 - x_1) + (y_2 - y)(y_2 - y_1) & (x_3 - x)(x_3 - x_1) + (y_3 - y)(y_3 - y_1) \end{vmatrix}.$$

It follows that Function `in_circle` can be implemented as the evaluation of the sign of a 2×2 determinant with $(2b + 3)$-bit integer entries.

A similar computation shows that the analogous Function `in_sphere` can be implemented as the evaluation of the sign of the following 3×3 determinant with $(2b+4)$-bit integer entries:

`in_sphere`(X)

$$= \frac{1}{x_1 - x} \begin{vmatrix} (x_1 - x)(y_2 - y) - (x_2 - x)(y_1 - y) & (x_1 - x)(y_3 - y) - (x_3 - x)(y_1 - y) & (x_1 - x)(y_4 - y) - (x_4 - x)(y_1 - y) \\ (x_1 - x)(z_2 - z) - (x_2 - x)(z_1 - z) & (x_1 - x)(z_3 - z) - (x_3 - x)(z_1 - z) & (x_1 - x)(z_4 - z) - (x_4 - x)(z_1 - z) \\ (x_2 - x)(x_2 - x_1) + (y_2 - y)(y_2 - y_1) + (z_2 - z)(z_2 - z_1) & (x_3 - x)(x_3 - x_1) + (y_3 - y)(y_3 - y_1) + (z_3 - z)(z_3 - z_1) & (x_4 - x)(x_4 - x_1) + y_4 - y)(y_4 - y_1) + (z_4 - z)(z_4 - z_1) \end{vmatrix}.$$

4.4. `intersections_sorting`. When constructing arrangements of line segments in the plane and trapezoidal maps (e.g., by a sweep-line algorithm), the following crucial numerical test is used. Let A_0A_1, A_2A_3, A_4A_5, and A_6A_7 be four line segments. The test consists in deciding if the x-coordinate x_I of the intersection point I of A_0A_1 and A_2A_3 is smaller or greater than the x-coordinate x_J of the intersection point J of A_4A_5 and A_6A_7. If the coordinates of points A_i are b-bit integers, this test reduces to evaluating the sign of a 2×2 determinant with $(3b + 3)$-bit integer entries.

Indeed, if $A_i = (x_i, y_i)$ for $i = 0, \ldots, 7$,

$$x_I = \frac{A_I}{B_I},$$

where

$$A_I = \begin{vmatrix} x_1 - x_0 & x_0 y_1 - x_1 y_0 \\ x_3 - x_2 & x_2 y_3 - x_3 y_2 \end{vmatrix},$$

$$B_I = \begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_3 - x_2 & y_3 - y_2 \end{vmatrix},$$

and a similar expression can be found for x_J. Comparing x_I and x_J reduces to testing the sign of the 2×2 determinant $\begin{vmatrix} A_I & B_I \\ A_J & B_J \end{vmatrix}$ where each element A_I and A_J are $(3b+3)$-bit integers while B_I and B_J are $(2b+3)$-bit integers.

5. Implementation and Experimental Results. Implementation has been done using C++ and the C++ ATT compiler on a Sun SS5-70. Times have been obtained using the `clock` command.

The entries are integers stored in a variable of type *double*. This allows us to manipulate exact 53-bit integers, to benefit the fast-floating point arithmetic of the processor and to handle overflows easily. For 3×3 determinants we do not apply exactly Theorem 2 and use a simplified algorithm that requires a $(b+2)$-bit arithmetic. More precisely, the algorithm does not look for the right order to compute $c(\lambda) = c(\lambda_1) + c(\lambda_2) + \upsilon_2$ as mentioned in Section 3.5.

The following table sums up the limits on the precision of the entries for the geometric tests of Section 4 using 32-bit, 53-bit, and 64-bit arithmetic:

		Line side	Plane side	In circle	In sphere	Intersection
		Determinant size				
Arithmetic	Function	2×2	3×3	2×2	3×3	2×2
32	Entries size	31	29	14	14	9
53	Entries size	52	50	24	23	16
64	Entries size	63	61	30	30	20

Implementation is available through WWW at url: `http://www.inria.fr:/prisme/personnel/devillers/anglais/determinant.html.`

The code has been tested on several kinds of determinants and compared with other methods that compute the determinant and subsequently test the sign.

Two variants of our method have been implemented.

STANDARD. The algorithm is iterated, until comparison of the entries and computation of the sign of minors in the three-dimensional case guarantees a conclusion. Evaluation of the sign of 2×2 determinants in the three-dimensional algorithm is computed using the lazy variant.

LAZY. The algorithm is combined with a floating-point filter. In the two-dimensional case the IEEE standard ensures that the sign of the determinant can be computed exactly when using the floating-point arithmetic as long as it is nonzero. Thus the exact arithmetic is invoked only in that case. In the 3D case, if the absolute value of the rounded determinant is bigger than 2^β where $\beta = \log \max \|x_i\| + \log \max \|y_i\| + \log \max \|z_i\| - b + 5$ we rely on the sign given by the rounded computation (IEEE norm ensure that the direct computation of the determinant D yields an approximate value $\tilde{D} = D(1 + 5\varepsilon_{\text{machine}})$. This filter is applied each time a new determinant is considered (i.e., each time some R' is substituted to U_3).

Direct Computation. This computation is done by first converting the original data from `double` to different numeric types and then the determinant is computed using the following formula:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \qquad \text{in the two-dimensional case,}$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = (a(ei - fh) + b(fg - di)) + c(dh - eg) \qquad \text{in the three-dimensional case.}$$

We have to mention that the type conversion between `double` and the other types used may be expensive and the time can be reduced if these types are used for the original data (this conversion time is included in the tables below).

The direct computation is run with different arithmetic.

DOUBLE. The computation done using `double` arithmetic is fast but not safe since it is subject to rounding errors.

QUADRUPLE. The computation is done using quadruple precision (`long double` type). For 2×2 determinants (but not for 3×3), this ensures exactness of the computation.

LEDA-INTEGER. LEDA provides exact computation on integers of arbitrary length. The result is exact [MN].

LN. LN Package, by S. Fortune and C. Van Wyk, works as a compiler, it transforms expression in LN language into optimized C++ routine computing the sign of the expression, first through a filter if it is not enough to conclude using optimized exact arithmetic (of fixed length). The result is exact [FV2].

Input. These methods have been tested and compared on a variety of inputs. We report here the most significant results.

We have run the above methods on null determinants, small determinants, and determinants of random matrices. In what follows a is said to be random on b bits if a is an integer evenly distributed in the range $-2^b + 1$ and $2^b - 1$.

Two-dimensional inputs

Random	:	$\begin{vmatrix} a & c \\ b & d \end{vmatrix}$	a, b, c, d random on 53 bits.
$x = -y$	:	$\begin{vmatrix} a & c \\ -a & -c \end{vmatrix}$	a, c random on 53 bits.
$x = -y + \varepsilon$	:	$\begin{vmatrix} a + e_a & c + e_c \\ -a + e_b & -c + e_d \end{vmatrix}$	a, c random on 53 bits, e_a, e_b, e_c, e_d on 2 bits.
$x = -y^t$	:	$\begin{vmatrix} a & -a \\ b & -b \end{vmatrix}$	a, b random on 53 bits.
$x = -y + \varepsilon^t$	:	$\begin{vmatrix} a + e_a & -a + e_c \\ b + e_b & -b + e_d \end{vmatrix}$	a, b random on 53 bits, e_a, e_b, e_c, e_d on 2 bits.
kU, lU	:	$\begin{vmatrix} ka & la \\ kb & lb \end{vmatrix}$	a, b random on 26 bits, k, l on 27 bits.
$kU, lU + \varepsilon$	:	$\begin{vmatrix} ka + e_a & la + e_c \\ kb + e_b & lb + e_d \end{vmatrix}$	a, b random on 26 bits, k, l on 27 bits, e_a, e_b, e_c, e_d on 2 bits.
$U, \lfloor \alpha U \rfloor$	:	$\begin{vmatrix} a & c \\ \lfloor \alpha a \rfloor & \lfloor \alpha c \rfloor \end{vmatrix}$	a, c random on 53 bits, α random in $[-1, 1]$.
$=$	:	$\begin{vmatrix} a & a \\ a & a \end{vmatrix}$	a random on 53 bits.
$= +\varepsilon$	:	$\begin{vmatrix} a + e_a & a + e_c \\ a + e_b & a + e_d \end{vmatrix}$	a random on 53 bits, e_a, e_b, e_c, e_d on 2 bits.

Running time (μs)

Input		Double	Quadruple	LEDA-integer	LN	Standard	Lazy
Random		0.58	165	47	2.1	2.39	0.66
Almost null	$x = -y + \varepsilon$	0.58	166	50	15.7	27.60	0.64
	$x = -y + \varepsilon^t$	0.56	164	47	15.6	3.49	1.41
	$kU, lU + \varepsilon$	0.52	152	47	15.9	24.91	0.66
	$U, \lfloor \alpha U \rfloor$	0.58	159	47	15.5	31.66	2.79
	$= +\varepsilon$	0.47	162	48	15.8	3.08	1.18
Null	kU, lU	0.49	148	47	15.9	26.26	32.10
	$=$	0.52	160	48	16.1	4.24	5.02
	$x = -y$	0.54	163	48	16.3	53.18	64.89
	$x = -y^t$	0.52	164	48	16.2	4.33	5.10

Three-dimensional inputs

Random	:	$\begin{vmatrix} a & d & g \\ b & e & h \\ c & f & i \end{vmatrix}$	$a, b, c, d, e, f, g, h, i$ random on 51 bits.
$x+y+z=0$	:	$\begin{vmatrix} a & d & g \\ b & e & h \\ -a-b & -d-e & -g-h \end{vmatrix}$	a, b, d, e, g, h random on 51 bits.
$x+y+z=0+\varepsilon$	:	$\begin{vmatrix} a+e_a & d+e_d & g+e_g \\ b+e_b & e+e_e & h+e_h \\ -a-b+e_c & -d-e+e_f & -g-h+e_i \end{vmatrix}$	a, b, d, e, g, h random on 51 bits, $e_a, e_b, e_c, e_d, e_e, e_f, e_g, e_h, e_i$ on 2 bits.
$x+y+z=0^t$	:	$\begin{vmatrix} a & d & -a-d \\ b & e & -b-e \\ c & f & -c-f \end{vmatrix}$	a, b, c, d, e, f random on 51 bits.
$x+y+z=0+\varepsilon^t$	:	$\begin{vmatrix} a+e_a & d+e_d & -a-d+e_g \\ b+e_b & e+e_e & -b-e+e_h \\ c+e_c & f+e_f & -c-f+e_i \end{vmatrix}$	a, b, c, d, e, f random on 51 bits, $e_a, e_b, e_c, e_d, e_e, e_f, e_g, e_h, e_i$ on 2 bits.
$kU, lV, mU+nV$	:	$\begin{vmatrix} ka & ld & ma+nd \\ kb & le & mb+ne \\ kc & lf & mc+nf \end{vmatrix}$	a, b, c, d, e, f, m, n random on 25 bits, k, l on 26 bits.
$kU, lV, mU+nV+\varepsilon$	:	$\begin{vmatrix} ka+e_a & ld+e_d & ma+nd+e_g \\ kb+e_b & le+e_e & mb+ne+e_h \\ kc+e_c & lf+e_f & mc+nf+e_i \end{vmatrix}$	a, b, c, d, e, f, m, n random on 25 bits, k, l on 26 bits, $e_a, e_b, e_c, e_d, e_e, e_f, e_g, e_h, e_i$ on 2 bits.
$U, V, \lfloor \alpha U + \beta V \rfloor$	:	$\begin{vmatrix} a & d & g \\ b & e & h \\ \lfloor \alpha a + \beta b \rfloor & \lfloor \alpha d + \beta e \rfloor & \lfloor \alpha g + \beta h \rfloor \end{vmatrix}$	a, b, d, e, g, h random on 51 bits, α, β random in $[-\frac{1}{2}, \frac{1}{2}]$.
$=$	:	$\begin{vmatrix} a & a & a \\ a & a & a \\ a & a & a \end{vmatrix}$	a random on 53 bits.
$=+\varepsilon$	:	$\begin{vmatrix} a+e_a & a+e_d & a+e_g \\ a+e_b & a+e_e & a+e_h \\ a+e_c & a+e_f & a+e_i \end{vmatrix}$	a random on 53 bits, $e_a, e_b, e_c, e_d, e_e, e_f, e_g, e_h, e_i$ on 2 bits

Running time (μs)

Input		Double	LEDA-integer	LN	Standard	Lazy
Random		1.9	225	3.0	14	3
Almost null	$x+y+z=0+\varepsilon$	2.0	217	65.5	310	7
	$x+y+z=0+\varepsilon^t$	2.0	222	65.4	134	11
	$kU, lV, mU+nV+\varepsilon$	2.0	215	65.9	288	15
	$U, V, \lfloor \alpha U + \beta V \rfloor$	2.0	215	65.3	303	37
	$=+\varepsilon$	2.2	209	71.0	164	185
Null	$kU, lV, mU+nV$	2.0	219	66.9	497	582
	$=$	2.0	188	77.5	18	23
	$x+y+z=0$	2.0	221	66.2	934	910
	$x+y+z=0^t$	2.0	219	65.7	38	39

The *standard* algorithm performs very well on determinants with random entries; even the nonlazy version acts as a filter. As this is the situation which is likely to be encountered in practice, our method is believed to run fast in most applications. The *null* cases are usually very uncommon in practice while the *almost null* cases are more realistic since nearly degenerate configurations are encountered. In such cases our performance is of the same order of magnitude as the *LEDA-integers*. Our algorithm is faster than LN in some cases and never exceeds LN by a factor 13.

6. Concluding Remarks. We have presented an algorithm that evaluates signs of 2×2 and 3×3 determinants with b-bit integer entries using only b and $(b+1)$-bit arithmetic, respectively. We have also shown how this algorithm can be used in several basic tests in geometric computation. The algorithm has been implemented and compared with the direct computation using several exact methods. Extensive experimental results have been given which demonstrate the efficiency of the algorithm.

An obvious direction for further research is to extend the present work to higher dimensions. The only difficulty there is to generalize Step 2 (Section 3.6) which reduces the last component by a constant factor while keeping the other components smaller than 2^b. Such an extension would provide, at least in principle, an extremely general solution to robustness in geometric computation since, by a result of Valiant [Val], any algebraic expression of size e can be constructively written as an $(e+2) \times (e+2)$ determinant whose entries are either variables or constants.

Acknowledgments. Jean-Pierre Merlet is acknowledged for supplying to us his interactive drawing preparation system JPdraw .

References

[BJMM] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. *Proc*. 5*th Canad. Conf. Comput. Geom*., pp. 73–78, Waterloo, 1993.

[Cla] K. L. Clarkson. Safe and effective determinant evaluation. *Proc*. 33*rd Annual IEEE Symp. Found. Comput. Sci*., pp. 387–395, 1992.

[FM] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. *Proc*. 7*th Annual ACM Symp. Comput. Geom*., pp. 334–341, 1991.

[For1] S. Fortune. Stable maintenance of point set triangulations in two dimensions. *Proc*. 30*th Annual IEEE Symp. Found. Comput. Sci*., pp. 494–505, 1989.

[For2] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. *Proc*. 8*th Annual ACM Symp. Comput Geom*., pp. 83–92, 1992.

[FV1] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. *Proc*. 9*th Annual ACM Symp. Comput. Geom*., pp. 163–172, 1993.

[FV2] S. Fortune and C. J. Van Wyk. Ln User Manual, 1994.

[GSS] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. *Proc*. 5*th Annual ACM Symp. Comput. Geom*., pp. 208–217, 1989.

[GY] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. *Proc*. 27*th Annual IEEE Symp. Found. Comput. Sci*., pp. 143–152, 1986.

[HHK] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl*., 9(6):50–59, November 1989.

[KLN] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans*. *Graph*., 10:71–91, 1991.

[Mil1] V. Milenkovic. Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic. Ph.d. thesis, Carnegie Mellon University, 1988.

[Mil2] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[Mil3] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. *Proc*. 30*th Annual IEEE Symp*. *Found*. *Comput*. *Sci*., pp. 500–505, 1989.

[MN] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Fachber. Inform., Univ. Saarlandes, Saarbrücken, 1989.

[SI] K. Sugihara and M. Iri. Geometric algorithms in finite-precision arithmetic. Technical Report 88-10, Math. Eng. and Physics Dept., Univ. of Tokyo, September 1988.

[Val] L. Valiant. Completeness classes in algebra. *Proc*. 11*th Annual ACM Symp*. *Theory Comput*., pp. 249–261, 1979.