Efficient Exact Evaluation of Signs of Determinants *

Hervé Brönnimann† Mariette Yvinec‡

Abstract: This paper presents a theoretical and experimental study on two different methods to evaluate the sign of a determinant with integer entries. The first one is a method based on the Gram-Schmidt orthogonalisation process which has been proposed by Clarkson [5]. We review the analysis of Clarkson and propose a variant of his method. The second method is an extension to $n \times n$ determinants of the ABDPY method [1] which works only for 2×2 and 3×3 determinants. Both methods compute the signs of a $n \times n$ determinants whose entries are integers on b bits, by using an exact arithmetic on only $b + O(n)$ bits. Furthermore, both methods are adaptive, dealing quickly with easy cases and resorting to the full-length computation only for null determinants.

Keywords: computational geometry, exact arithmetic, precision, robust algorithms.

1 Introduction

Geometric algorithms are known to be highly sensitive to numerical inaccuracy. Those algorithms generally rely on building discrete combinatorial structures whose actual state depend on the outcomes of some numerical tests. In this context, roundoff errors leads quickly to fatal inconsistencies and failure of the programs. Robustness has now become one of the major issue in the field of computational geometry [4, chap. 10].

Some attempts have been made to design geometric algorithms such that robust implementations can be obtained using only the inaccurate but fast arithmetic provided by floating point processors (see for examples [17, 18, 15, 13, 10, 11, 8]). Such solutions, although very useful in some domains like solid modeling and CSG applications, are still painful to design and known only for a few geometric problems. Another approach is to turn to exact arithmetic which makes robustness a non issue. The use of exact arithmetic has been recently advocated by Fortune and Van Wyk [9], Yap [19, 20] , Burnikel and coll. [3] and many others. However, as reported for example by Karasick and coll. [12], naive implementation of exact arithmetic can be quite slow and many works are now devoted to speed up the paradigm of exact geometric computing. Fortunately exact geometric computing does not imply computing everything exactly. Most of the numerical tests arising in geometric algorithm amount to determine only the sign of a determinant or of a polynomial expression. Thus arithmetic filters based on a fast floating point evaluation of the expression and of a bound on the error allow very often to make safe decision. This approach is used for example in the LN package by Fortune and Van Wyk [9] and has been shown experimentally to provide a substantial speed-up. Devillers and Preparata investigate the theoretical behavior of some filters [6]. In degenerate or near degenerate cases, however, exact arithmetic has to be carried out in full. Burnikel and coll. [3] and Yap [19] provide powerful software to perform exact arithmetic on algebraic numbers. But numerical tests arising in geometric algorithms are not arbitrary. In fact, most geometric algorithms rely on a few number of geometric predicates such as which-side or orientation test, in-circle or in-sphere tests, all of which amount to compute the sign of a determinant. Thus designing a specialized implementation for evaluating exactly the sign of a determinant can in many cases avoid to pay the price of a general purpose multi-precision package. Clarkson [5]

*This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

†INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis cedex (France).

‡INRIA Sophia-Antipolis and CNRS, Laboratoire I3S, BP 145, 06903 Sophia-Antipolis cedex (France)

To appear in 13th Annual ACM Symposium on Computational Geometry, June 4–6 1997, Nice, France.

propose an efficient method to compute the sign of a determinant whose entries are integers. The so called ABDPY method, due to Avnaim and col. [1], is an alternative solution for 2×2 and 3×3 determinants. Using an uncommon multiple-term arithmetic, Schewchuck [16] designs an adaptative implementation for low dimensional geometric predicates on floating point entries.

In this paper, we propose a theoretical and experimental study on the exact evaluation of the sign of a determinant with integer entries. Mainly, we have revisited the method of Clarkson and extended to higher dimensions the ABDPY method.

Clarkson's method is based on the Gram-Schmidt orthogonalization process and will be called hereafter the reorthogonalization method. We propose a variant of his algorithm which is somewhat simpler to analyze. This variant allows to compute the sign of an $n \times n$ determinant whose entries are b-bit integers, that is, integers whose absolute values are smaller than 2^b, using an exact arithmetic on $b + \lceil 1.32(n-1) + 0.5 \log n \rceil$ bits, and with a worst case complexity of $O(bn^3 + n^3 \log n)$.

We call lattice method the extension of ABDPY method to higher dimensions. In fact the lattice method, borrows from both the ABDPY method and Clarkson's method. Like ABDPY, it mostly consists in locating one of the column vectors of the determinant with respect to a region approximating the hyperplane spanned by the others column vectors. But, when the vector is found to lie within the approximating region, the algorithm resorts to an iterative doubling technique which was suggested to us by the study of Clarkson's method. For a $n \times n$ determinant with b-bit integer entries the lattice method requires an exact arithmetic on $b + n - 2 + \lceil \log n \rceil$ bits and, though its worst case complexity is exponential, its behavior is $O(bn^3)$ in most cases.

The efficiency of both methods comes mostly from the small number of extra bits they require. Indeed, in most practical cases, the exact arithmetic required will stay within the 53 bits of precision available in the mantissa of standard IEEE doubles. Then all numerical computations entailed by the evaluation of the sign of a determinant can be performed exactly using the floating point processor of any computer conforming to the IEEE norm. Another factor of efficiency of those methods is their adaptivity. Both are iterative methods and the number of iterations performed depends on the actual value of the determinant. This number is quite small in easy cases where the determinant is far from zero.

The next two sections respectively present the reorthogonalization and the lattice method. Experimental evidence of their efficiency is given in the last section.

2 The reorthogonalization method

Overview. The goal is to compute $\det(\mathcal{A})$, where $\mathcal{A}$ is a matrix whose coefficients are b-bit integers. One may use Gram-Schmidt reductions to compute an orthogonal matrix $\mathcal{C}$ whose determinant is the same as that of $\mathcal{A}$, and such that each column C_k of $\mathcal{C}$ is $A_k + \mathrm{LC}(A_1, \ldots, A_{k-1})$, where LC denotes any linear combination of its arguments. Computing the determinant of $\mathcal{C}$ can then be done by standard Gaussian elimination. In practice, roundoff errors result in a matrix $\mathcal{B}$ that approximates $\mathcal{C}$. The sign of $\det(\mathcal{B})$ is correct if $\mathcal{A}$ is well-conditioned. It could be wrong, however, if $\mathcal{A}$ is ill-conditioned. Clarkson's idea amounts to preconditioning the matrix $\mathcal{A}$ without introducing errors, before doing Gram-Schmidt reductions. The invariant enforced by preconditioning is that each column B_k of $\mathcal{B}$ obtained by reducing the preconditioned A_k verifies ${2B_k}^2 \geq {A_k}^2$.

Clarkson's preconditioning applies a variant of Gram-Schmidt reductions in which the vector subtracted from A_k is a linear combination of $A_k^{(1)}, \ldots, A_{k-1}$ (rather than $C_k^{(1)}, \ldots, C_{k-1}$ for standard Gram-Schmidt reductions). Thus, whatever the coefficients in this combinations be, $\det(\mathcal{A})$ is not affected as long as the linear combination is computed exactly. Nevertheless, the norm of A_k decreases in the process, so we may multiply it by a suitable factor $s \geq 2$ without overflowing. Iterating for a given k until the invariant is satisfied leads to a transformed matrix $\mathcal{A}'$ whose determinant has been multiplied by a factor $\prod s > 0$. Therefore the sign of $\det(\mathcal{A}')$ is the same as that of $\det(\mathcal{A})$.

The iterative preconditioning process is depicted in figure 1, and the code for preconditioning and reducing is given below. It is the same as that of [5]. Our variant is in the choice of s that leads to a much simpler analysis.

1. **for** $k := 1$ **to** n
2. **loop**
3. $B_k^{(k)} := A_k$
4. **for** $j := k-1$ **downto** 1
5. $B_k^{(j)} := fl\left(B_k^{(j+1)} - fl\left(fl\left(\frac{A_k}{B_j}\right) B_j\right)\right)$
6. **if** $fl\,(A_k \cdot A_k) \leq 2fl\left(B_k^{(1)} \cdot B_k^{(1)}\right)$
7. $B_k := B_k^{(1)}$, **exit loop**
8. Compute some integer s
9. $A_k^{(k)} := s \times A_k$
10. **for** $j := k-1$ **downto** 1
11. $A_k^{(j)} := A_k^{(j+1)} - \left\lceil fl\left(\frac{A_k^{(j+1)}}{B_j}\right)\right\rfloor A_j$
12. $A_k := A_k^{(1)}$
13. **end loop**

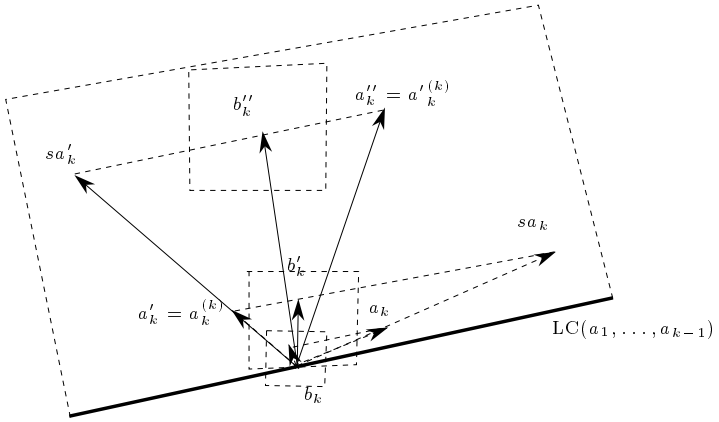

Figure 1: The process of reducing vector A_k is shown. At any time, A_k remains in the bounding box, but its component along the orthogonal of $(A_1, \ldots, A_{k-1})$ increases at each step. The computed vector B_k lies in the square box. At the first and second steps, the box does not ensure that the sign of the determinant is known safely, but it becomes so after the third reduction of A_k.

Notation *fl* (expr) means to evaluate an expression within machine floating point precision. Lines 3–5 compute the Gram-Schmidt reductions, lines 6–7 check if the invariant is satisfied, and if not, lines 10–13 compute the modified Gram-Schmidt reductions and loop.

When the determinant is null, some vector C_k is null and the invariant $2\|B_k\|^2 \geq \|A_k\|^2$ is never satisfied. In this case, the algorithms stops and concludes that the determinant is null after a certain number of iterations have been performed (this number is given in the complexity analysis below). Otherwise, let $\mathcal{B}'$ be the floating point approximation of $\mathcal{C}'$, obtained from $\mathcal{A}'$ by Gram-Schmidt reductions. Since the preconditioning invariant is satisfied, it is not hard to show that $\det(\mathcal{B}')$ can be evaluated with relative precision less than 1 (using standard Gaussian elimination [5, 7]). Hence the sign of $\det(\mathcal{A})$ is computed exactly, and a good approximation $(\prod s)^{-1} \det(\mathcal{B}')$ is available.

Needed arithmetic. To bound the number of bits needed by the algorithm, we first obtain a bound on the error $\mathcal{B} - \mathcal{C}$ and deduce a bound on the norms of the coefficients of $\mathcal{A}$ after several reduction steps.

To bound the error $\mathcal{B} - \mathcal{C}$, we assume that the algorithm has conditioned $A_1, \ldots, A_{k-1}$ (therefore the preconditioning invariant is valid for $j < k$), and has performed a number of preconditioning steps on A_k. When we exit the reduction of A_k, the invariant is satisfied for $j = k$. In [5], it is shown that this implies that $\|B_j - C_j\| \leq \delta_j \|B_j\|$ for all $j \leq k$, where the δ_j's depend on j, n, and the precision $\mathbf{u}$ of the floating point computations. We can assume that $\delta_n \leq 0.01$, for all practical values of n and $\mathbf{u}$, which implies loose limits on n as a function of $\mathbf{u}$.

Having bounded the error $\mathcal{B} - \mathcal{C}$, we must now bound the coefficients of $\mathcal{A}$ after several reduction steps. Let A'_k be the vector A_k after several modified Gram-Schmidt reductions. As is done in [5], we can bound the norms of all the vectors $A_k^{(j)}$ computed in line 11 by

$$\left\| A_k^{(j)} \right\|^2 \leq 0.55\, s^2 \|A'_k\|^2 + 0.51\, S_k^c\,, \qquad (1)$$

where $S_k^c = \sum_{j=1}^{k-1} \|C_j\|^2$. It is now clear that, to control the growth of A_k after several reductions, we must choose s carefully in terms of S_k^c and $\|A'_k\|$. The value S_k^c is unknown to the algorithm, however, and only an approximation S_k^b of S_k^c is known. Assuming that $\delta_n \leq 0.01$, we can show (as in done in [5]) that $\left|S_k^b - S_k^c\right| \leq 0.07 S_k^c$.

Clarkson's choice [5] is given in terms of S_k^b and $\|B_k\|$, and yields a complicated analysis. Instead, we pick (with a parameter λ to be set later)

$$s = \left\lceil fl\left(\sqrt{1.3 + \frac{S_k^b}{\lambda \left\|A'_k\right\|^2}}\right)\right\rfloor.$$

If $s = 1$, we must ensure that A'_k is indeed shrinking, otherwise the algorithm could loop infinitely. But if $s = 1$, the square root is (conservatively) smaller than 1.51, hence $S_k^b \geq \lambda \|A'_k\|^2$, and we know that $S_k^c \leq 1.07 S_k^b$. Plugging into (1) shows that $\left\|A_k^{(1)}\right\|^2 \leq 0.57(1+\lambda)\|A'_k\|^2$. We now set $\lambda = 0.42$ to obtain $\left\|A_k^{(1)}\right\| \leq 0.9\|A'_k\|$. Thus, if $s = 1$, A'_k shrinks in norm by at least 10%.

Otherwise, put $s' = 1.3 + S_k^b/(\lambda\|A'_k\|^2)$ so that $s = \lceil fl\,(s')\rfloor$. Estimating rounding errors conservatively, we can safely say that s' is less than $(s - 0.51)^2$, and hence that

$$s^2\|A'_k\|^2 \leq \frac{s^2}{(s-0.51)^2 - 1.3}\frac{1.07}{\lambda} S_k^c \leq 11.05, S_k^c, \qquad (2)$$

since s is greater than 2 and λ is set at 0.42. Plugging this bound into (1), we finally get that $\left\|A_k^{(1)}\right\|^2 \leq 6.59\, S_k^c$, which bounds the norm of the next vector A'_k.

From all this, we gather (conservatively) that

$$\|A'_k\|^2 \leq \max\left(\|A_k\|^2, 6.59 S_k^c\right)$$

where A'_k is after any number of reductions of A_k. The key here is that S_k^c depends only $C_1 \ldots, C_{k-1}$ and remains fixed during the k-th reduction loop (no matter how many times we enter the loop).

To obtain a bound on how many bits are needed by the algorithm is now routine. If $\mathcal{A}$ has b-bit integer

coefficients, the maximum norm of a vector A_j is $\sqrt{n}2^b$. Then $\left\|A_k^{(j)}\right\|^2 \leq \max\left(n2^{2b}, 6.59S_k^c\right)$ holds for all $k \geq n$, for all $j \leq k$, and after any number of reductions of A_k. By induction, we find that $S_k^c \leq n\,6.59^{k-2}2^{2b}$. Hence $b + \lceil 1.36(n-1) + 0.5\log n\rceil$ bits suffice to express all the vectors A_k and $A_k^{(j)}$ occurring in the algorithm.

Complexity. It is also simple to bound the number of iterations. Indeed, each iteration in which $s = 1$ decreases the norm of A'_k by 10% while keeping the determinant constant, and the others multiply the determinant by a value $s \geq 2$ while keeping the coefficients of $\mathcal{A}$ in the allowable range. When the determinant is not null, each iteration of the first kind decreases the orthogonality defect $\prod_{1\leq j\leq k}\|A_j\|/\|C_j\|$ by 10%, and the defect is not increased by iterations of the second kind; since it is greater than 1 and smaller than $\det(\mathcal{A})$, there cannot be more than $t_1 = O(\log\det(\mathcal{A}))$ such iterations. After t_2 iterations of the second kind, we have $2^{t_2}|\det(\mathcal{A})| \leq |\det(\mathcal{A}')| \leq n^{n/2}2^{bn}$. Hence $t_1 + t_2 = O(bn + n\log n)$. When the determinant is null, the algorithms stops after $O(bn+n\log n)$ iterations and concludes that the determinant is null. Each iteration performs $O(n^2)$ operations, and the final Gaussian elimination performs $O(n^3)$ operations. Hence the complexity of the algorithm is $O(bn^3 + n^3\log n)$.

In fact, the algorithm does not usually perform so many iterations. When reducing A_k, the product $P_k^c = \prod_{1\leq j\leq k}\|C_j\|$ has been multiplied by a quantity which we denote by $\prod s$. Since P_k^c is either 0 or greater than 1 to start with, it must be zero or at least $\prod s$. The following upper bound on P_k^c can be computed by the algorithm:

$$P_k^b = \prod_{j=1}^{k-1}(1+\delta_j^2)\|B_j\|^2 \times (\|B_k\|^2 + \delta_k^2\|A'_k\|^2).$$

As soon as $P_k^b \leq \prod s$, we can rest assured that the first k columns are linearly dependent and that the determinant is 0. Since $\mathcal{B}'$ is close in practice to $\mathcal{C}'$, this is usually how the loop is exited when the determinant is null.

3 The lattice method

Overview. Here and in the following we assume that we want to evaluate the sign of the determinant $D = \det(U_1, U_2, \ldots, U_n)$ where $U_1, U_2, \ldots, U_n$ are n-dimensional column vectors whose coordinates are supposed to be b-bit integers. The lattice method considers in a special way the last column and last row of determinant D. We note $z_1, z_2 \ldots z_n$ the last components of the input vectors and $u_1, u_2, \ldots, u_n$ their orthogonal projections on $\mathbb{R}^{n-1}$. Without loss of generality, $z_1, z_2, \ldots, z_n$ are assumed to be non negative.

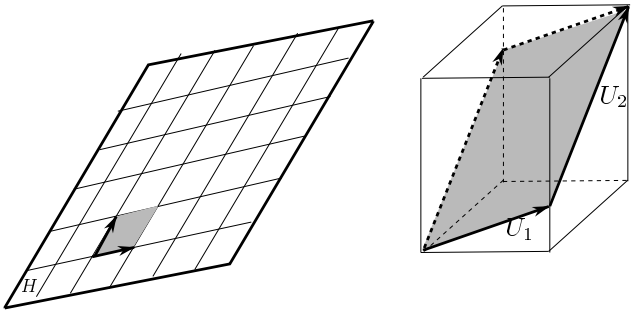

Figure 2: The cell $\mathcal{C}_H$ and the box $\mathcal{B}$

Let H be the hyperplane through $\{O, U_1, U_2, \ldots, U_{n-1}\}$. The basic underlying idea is that computing the sign of the $n \times n$ determinant D reduces to compute the sign of a $(n-1)\times(n-1)$ determinant if the position of the point U_n with respect to the hyperplane H is known. Indeed, assuming that vectors $U_1, U_2, \ldots, U_{n-1}$ together with E_n, the unit vector along the nth axis, span $\mathbb{R}^n$, we have $U_n = \mathrm{LC}(U_1, U_2, \ldots, U_{n-1}) + \alpha E_n$ and $D = \alpha\det(u_1, u_2, \ldots, u_{n-1})$ where the sign of α only depends of the position of U_n with respect to the hyperplane H.

Locating U_n with respect to H amounts in turn to evaluate the sign of D, and this might look like going round in circle. In fact, in a first phase, point U_n is located with respect to some region $\mathcal{H}$ of $\mathbb{R}^n$ which contains H and can be considered as an approximation of H. If U_n is found to be outside region $\mathcal{H}$, the position of U_n with respect to H is known and the problem is reduced by one dimension. Otherwise (and this is where the lattice method mostly departs from ABDPY), the algorithm enters a second phase in which the last column vector and therefore the numerical value of the determinant are iteratively doubled, which amounts to iteratively refine the approximation $\mathcal{H}$ of H.

Before being more precise on those two phases of the algorithm, let us describe region $\mathcal{H}$. We consider the lattice $\mathcal{L}_H$ formed by vectors in H that are linear combination of $\{U_1, U_2, \ldots, U_{n-1}\}$ with integer coefficients, and the lattice $\mathcal{L}$ which is the projection of $\mathcal{L}_H$ in $\mathbb{R}^{n-1}$.

$$\mathcal{L}_H = \left\{\sum_{i=1}^{n-1} l_i U_i, l_i \in \mathbb{N}\right\}, \quad \mathcal{L} = \left\{\sum_{i=1}^{n-1} l_i u_i, l_i \in \mathbb{N}\right\}.$$

The lattice $\mathcal{L}$ induces a partition of $\mathbb{R}^{n-1}$ into elementary cells each of which is a translated copy of the origin cell : $\mathcal{C} = u_1 \oplus u_2 \oplus \ldots \oplus u_{n-1} = \{\sum_{i=1}^{n-1}\alpha_i u_i,\ \ 0 \leq \alpha_i < 1\}$, where the $\oplus$ symbol stands for a (half-closed) Minkowski sum. We note $\mathcal{C}(l_1, \ldots l_{d-1})$ the lattice cell that is the translated of $\mathcal{C}$ by the vector $\sum_{i=1}^{d-1} l_i u_i$. The point $c(l_1, \ldots l_{d-1}) = \sum_{i=1}^{d-1} l_i u_i$ is called the reference

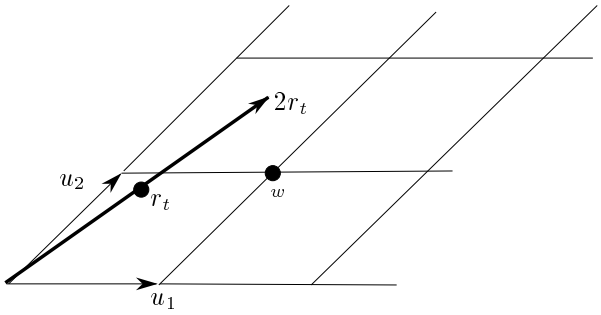

Figure 3: Locating $2r_t$ in the lattice $\mathcal{L}$

point of the cell $\mathcal{C}(l_1, \dots l_{d-1})$. Analogous definitions and notation hold for the lattice $\mathcal{L}_H$ whose elementary cells partition H. The elementary cell $\mathcal{C}(l_1, \dots l_{d-1})$ of $\mathcal{L}$ is the orthogonal projection of the cell $\mathcal{C}_H(l_1, \dots l_{d-1})$ of $\mathcal{L}_H$.

To describe the region $\mathcal{H}$, we consider the n-dimensional box (see figure 2)

$$\mathcal{B} = u_1 \oplus u_2 \oplus \dots \oplus u_{n-1} \oplus \left(\sum_{i=1}^{n-1} z_i \right) E_n.$$

Box $\mathcal{B}$ has the origin cell $\mathcal{C}$ of lattice $\mathcal{L}$ as orthogonal projection and containss the origin cell $\mathcal{C}_H$ of $\mathcal{L}_H$. Then, region $\mathcal{H}$ is defined as the union of all the boxes $\mathcal{B}(l_1, \dots l_{n-1})$ which are copies of $\mathcal{B}$ translated by the vectors of lattice $\mathcal{L}_H$. Since $\mathcal{B}$ icontains the origin cell $\mathcal{C}_H$, $\mathcal{B}(l_1, \dots l_{n-1})$ contains the cell $\mathcal{C}_H(l_1, \dots l_{d-1})$ and $\mathcal{H}$ contains H.

Now, the overview of the algorithm is a follows.

• *First phase*. The algorithm determines the position of point U_n with respect to $\mathcal{H}$.

- If point U_n is found to be above or below $\mathcal{H}$, the relative position of U_n with respect to H is known and the algorithm ends up with evaluating the sign of a $(n-1) \times (n-1)$ determinant.
- Otherwise, U_n lies within some box $\mathcal{B}(k_1, \dots k_{n-1})$ of $\mathcal{H}$ and the algorithm computes the vector $R = U_n - \sum_{i=1}^{n-1} k_i U_i$ which satisfies both

$$D = \det(U_1, U_2, \dots, R)$$

and $R \in \mathcal{B}$. Then, the second phase is entered.

• *Second phase*. A variable number of the following iterations are performed. Iteration t takes as input a vector R_t of $\mathbb{R}^n$ such that

$$D_t = \det(U_1, U_2, \dots, R_t) = 2^t D$$

and $R_t \in \mathcal{B}$. (Initially, $D_0 = D$ and $R_0 = R$.) The algorithm sets $R' = 2R_t$ and considers the determinant $D_{t+1} = \det(U_1, U_2, \dots, R') = 2D_t$. Point R' is located with respect to the union of boxes $\mathcal{H}$.

- If R' is found outside $\mathcal{H}$ the relative position of R' and H is known and the algorithm ends up getting the sign of D_{t+1} which is also the sign of D from the sign of a $(n-1) \times (n-1)$ determinant.
- Otherwise R' belongs to a box $\mathcal{B}(k_1, \dots k_{n-1})$ of $\mathcal{H}$ and the algorithm computes the vector $R_{t+1} = R' - \sum_{i=1}^{n-1} k_i U_i$ which verifies both

$$D_{t+1} = \det(U_1, U_2, \dots, R_{t+1}) = 2^{t+1} D$$

and $R_{t+1} \in \mathcal{B}$. Then one proceeds to the next iteration with D_{t+1}.

The algorithm ends up evaluating the sign of a $(n-1) \times (n-1)$ determinant as soon as one of the points $R' = 2R_t$ is located outside $\mathcal{H}$. If the determinant D is null, this will never happen but the algorithm can stop and be sure than D is zero after at most $bn + \lceil (\frac{n}{2} + 1) \rceil \log n$ iterations. Indeed, at each iteration the value of the determinant is multiplied by a factor 2. Thus, $D_t = 2^t D$ which is more than 2^t if D is not zero. On the other end, each entry of determinant D_t is at most 2^b except those of the last column (the components of R_t) which are at most $n2^b$ because R_t belongs to box $\mathcal{B}$. Thus $D(t)$ is less than $n\sqrt{n}^n 2^{bn}$. Therefore, when $2^t > n\sqrt{n}^n 2^{bn}$, the algorithm can stop and conclude that D is null.

Further details. Due to lack of space, we skip the detailed description of the first phase.

At each iteration of the second phase, the algorithm is given a point $R_t(r_t, z_t)$ in $\mathcal{B}$ and has to locate the endpoint of $R' = 2R_t$ with respect to $\mathcal{H}$. For this the algorithm determines the reference point c_{t+1} of the cell of lattice $\mathcal{L}$ containing the endpoint of the projection $2r_t$ of R'. Owing to the fact that r_t belongs to the origin cell $\mathcal{C}$, point c_{t+1} is necessarily one of the 2^{d-1} vertices of $\mathcal{C}$ (see figure 3) and finding which one amounts to determine the sign of the following $(n-1) \times (n-1)$ determinants, for $i = 1, \dots, n-1$:

$$a_i = [u_1, \dots, u_{i-1}, (2r_t - w), u_{i+1}, \dots, u_{n-1}],$$

where $w = \sum_{i=1}^{n-1} u_i$. Let C_{t+1} be the lattice point of H that projects on c_{t+1}. We compute $R_{t+1} = 2R_t - C_{t+1}$. If the last component z_{t+1} of R_{t+1} is negative or greater than $\sum_{i=1}^{n-1} z_i$, the position of R_{t+1} with respect to H is known. Else $D_{t+1} = \det(U_1, U_2, \dots, R_{t+1}) = 2^{t+1} D$ and $R_{t+1} \in \mathcal{B}$ and the algorithm proceeds to iteration $t+1$.

Needed arithmetic. To evaluate the sign of an $n \times n$-determinant, the above algorithm performs different operations among which are the evaluations of signs of $(n-1) \times (n-1)$-determinants. To evaluate the signs of those determinants, the algorithm calls the $(n-1)$-dimensional version of the same algorithm which in turn involves evaluating signs of $(n-2) \times (n-2)$ determinants and so on until dimension 2 is reached, where the original ABDPY method is used. We call k-th level

Test	Det3x3			Det4x4			Det5x5			Det6x6		
	$b = 50$			$b = 49$			$b = 47$			$b = 46$		
Determinant	R	Q	N	R	Q	N	R	Q	N	R	Q	N
Gauss	9	9	9	13	13	12	23	21	22	36	35	34
Leda	339	334	337	1661	1648	1650	5503	5487	5250	9800	8600	8300
Reorth.	4	30	403	77	386	1100	174	623	2211	600	2140	7400
Lattice	11	25	345	88	187	1232	315	704	2460	na	na	na
Filt.+Lat.	3	13	355	13	104	1171	46	565	2453			
LN	3	53	53	4	295	296	na	na	na			

Figure 4: Timings in microseconds on a Sun Sparc5, 110MHz, running Solaris (na: not available). R, Q, N stand respectively for random, quasi-null, and null determinants

of computation the set of numerical computations performed in calls to the algorithm evaluating the sign of $k \times k$-determinants except the computations involved in evaluating the sign of $(k-1) \times (k-1)$-determinants. In the following, we first focus on the computations performed at the n-th level and then consider the lower levels.

At the n-th level of computations, the input vectors $\{U_1, U_2, \ldots U_n\}$ are all vectors with b-bit integer entries. However to be able to generalize our conclusions to lower levels, we shall here assume a weaker hypothesis. Namely, we consider the L_∞ norm of the input vectors (the L_∞ norm $\|U\|_\infty$ of a vector U is the maximum absolute value of any components of U), and define the *norm* of determinant $\det(U_1, U_2, \ldots U_n)$ as the sum $\sum_{i=1}^{d} \|U_i\|_\infty$ of the L_∞ norm of its column vectors. The following fact is proved in the full length version of this paper.

Fact 1 *If the norm of D is less than a constant S_n, any of the $(n-1)\times(n-1)$ determinants considered by the algorithm has a norm less than $S_{n-1} = 2S_n$. Furthermore computations performed at the nth level do not require exact integer arithmetic on more than $\lceil \log S_n \rceil + 1$ bits.*

An easy recurrence using the above fact shows that if the norm of the $n \times n$ determinant D is less than S_n, the computations performed at level k require an exact integer arithmetic on $\lceil \log S_k \rceil + 1$ bits with $S_k = 2^{n-k} S_n$. For a $n \times n$-determinant with b-bits integer entries, the norm hypothesis is satisfied with $S_n = n2^b$. Since the original ABDPY algorithm yields the sign of a 2×2 determinant with b'-bits integer entries using only a b'-bit arithmetic [2], the whole algorithm requires only an exact integer arithmetic on $n - 2 + \lceil \log n \rceil + b$ bits.

Complexity. During the first phase, the algorithm requires at most $O(bn)$ evaluations of signs of $(n-1)\times(n-1)$ determinants (this is clear from the full description of the first phase of the algorithm). During the second phase, the algorithm performs at most $O(bn + n \log n)$ iterations each of which involves $(n-1)$ evaluation of signs of $(n-1) \times (n-1)$ determinants. The other computations at the n-th level take time $O(n^2)$. Thus the complexity t_n obeys a recurrence equation of the form

$$t_n = O(bn^2 + n^2 \log n) t_{n-1} + O(n^2),$$

which leads to an exponential complexity $t_n = O((b + \log n)^{n-1}(n!)^2)$. But this bound is very pessimistic: indeed, even if the determinant is null and requires a full-fledged first phase followed by bn iterations in the second phase, the $(n-1) \times (n-1)$ encountered determinants have no reason to be close to zero. Therefore they are very likely to be caught by some floating point filter. As the different determinants arising in the loops of the first phases or during iterations of the second phase differ only by a single column, the floating point evaluations involved in filtering are reduced to a scalar product (using the minors relative to the changing column). Thus in practice, evaluating the sign of each encountered $(n-1) \times (n-1)$ determinant takes only $O(n)$ time and the time required by the whole algorithm should be close to $O(bn^3 + n^3 \log n)$ even for a null determinant. Of course, if the determinant is not even close to zero, few iterations are performed and the algorithm is very fast.

4 Experimental results

The reorthogonalization method and the lattice method have both been implemented in C. The lattice method is at present time available up to dimension five only but will be soon available in dimension 6 and higher. To be able to compare the efficiency of those methods with respect to others we have also implemented a floating point Gaussian elimination (which of course does not always yields the right sign) and an exact computation of the determinant using the exact integer arithmetic provided by LEDA. In order to show the practical efficiency of our methods and to get a fair comparison with the code produced by LN [9] for sign of determinants computation, we have also implemented the lattice method combined with a floating point filter (the results would have been quite similar for the reorthogonalization method).

We have experimented on determinants of dimensions n from 2 to 6 with b-bit integers entries, where

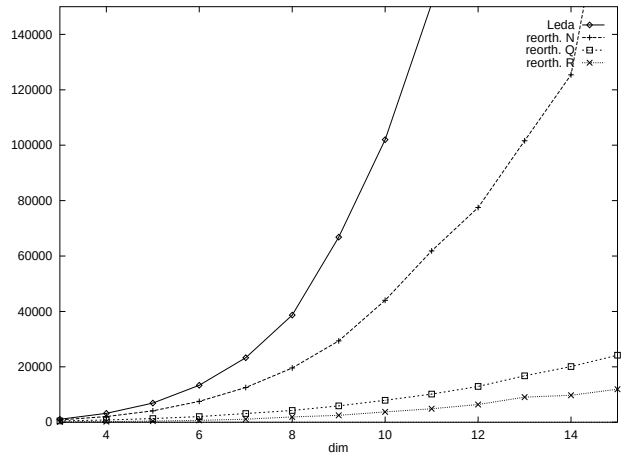

Figure 5: Timing as function of the dimension for the exact computation of signs of $n \times n$ determinants. The timing are in microsecond. The solid curve correspond respectively to an exact computation using Leda. The three dotted curves correspond to the reorthogonalization method on respectively null, quasi-null and random determinants. Using the lattice method instead would yield almost identical behavior.

$b = 53-(n-2+\lceil \log n \rceil)$. (This is the precision bound allowed for the lattice method and slightly over the bound allowed for reorthogonalization, however these bounds are pessimistic and, in practice the reorthogonalization method works and yields the correct sign up to this precision on the entries). In each dimension, three types of determinants respectively called *random*, *null* and *quasi-null* have been used. Random determinants have as entries random signed integers numbers on b bits and their value is of the the order of 2^{bn}. Null determinants are formed by $n-1$ column vectors of the form $k_i U_i$ and a last column of the form $\sum_{i=1}^{n-1} l_i U_i$ where the components of vectors U_i are random signed integers numbers on $\lceil b/2 \rceil$ bits while the coefficients k_i and l_i are random signed integers numbers on $\lfloor b/2 \rfloor$ bits. Therefore null determinants have rank $n-1$. Quasi-null determinants are obtained by a small perturbation of null determinants adding to each entry a random sign integer on two bits. Timing results appear in figures 4, 4.

Lattice and reorthogonalization method show about the same performances at least up to dimension 6. Both appear to be highly adaptive method being very fast for random determinants (less than a factor 10 above the floating point calculation) and also fast enough for quasi-null determinants which are not caught by usual floating point filters. Both method are still faster than the exact computation for null determinants. Furthermore the speeding factor indeed increases as the dimension rise. This is confirmed by tests performed on the reorthogonalization method up to dimension 12 and whose results are gathered in the graph of figure 4. At least for dimension n up to 6, the timing results for null determinants agree with the predicted $O(n^3)$ law. It is to be noted however that the lattice method can be very slow in some very special cases such that when all the entries of the determinant are nearly equal (this is in fact the only case we have experimentally found) This is in agreement with the above analysis because in such case every lower dimensional determinant considered by the algorithm are close to zero. Another caveat to be mentioned : any of the above method except Leda will fail in dimension n equal to 20 (possibly less) due to overflow of the range of exponents in IEEE double precision. At last, we can also mention that reorthogonalization in dimension 6 has been successfully used to detect the singular configurations of a parallel robot with six degrees of freedom an application where lots of null and quasi-null determinants have to be tested.

5 Conclusions

Both reorthogonalization and lattice method appear to be practically feasible methods to compute exactly signs of determinants with integer entries at least for dimension up to 15. This should provided an affordable way of performing exact tests in most geometric algorithms which is a decisive step toward robustness.

References

[1] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. Research Report 2306, INRIA, BP93, 06902 Sophia-Antipolis, France, 1994.

[2] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C16–C17, 1995.

[3] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[4] B. Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., April 1996.

[5] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.

[6] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. Rap-

port de recherche 2971, INRIA, 1996. also report CS96-27 Brown University.

[7] G. Forsythe and C. Moler. *Computer solutions of linear algebraic systems.* Prentice Hall, 1967.

[8] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

[9] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[10] C. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[11] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 106–117, 1988.

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

[13] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 235–243, 1990.

[14] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[15] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[16] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[17] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inform. Proc.*, 12(4):380–393, 1989.

[18] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4:179–228, 1994.

[19] C. K. Yap. Towards exact geometric computation. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 405–419, 1993.

[20] C. K. Yap and T. Dubhe. The exact computation paradigm. In D. Du and F. Hwang, editors, *Computing in Euclidean Geometry.* World Scientific Press, 1995.