Efficient Exact Evaluation of Signs of Determinants

Hervé Brönnimann, Mariette Yvinec

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Exact Evaluation of Signs of Determinants

Hervé Brönnimann , Mariette Yvinec

N° 3140

Mars 1997

THÈME 2

ISSN 0249-6399

Efficient Exact Evaluation of Signs of Determinants

Hervé Brönnimann[*] , Mariette Yvinec[**]

Thème 2 — Génie logiciel
et calcul symbolique
Projet Prisme

Rapport de recherche n° 3140 — Mars 1997 — 26 pages

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. 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 which works only for 2×2 and 3×3 determinants. Both methods compute the signs of a $n \times n$ determinant 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.

Key-words: Computational geometry, exact arithmetic, precision, robust algorithms

(Résumé : tsvp)

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

[*] INRIA Sophia-Antipolis, Phone : +33 (0) 4 93 65 77 75, Email : Herve.Bronnimann@sophia.inria.fr
[**] INRIA Sophia-Antipolis and CNRS, Laboratoire I3S, BP 145, 06903 Sophia-Antipolis cedex (France), Phone : +33 (0) 4 93 65 77 49, Email :Mariette.Yvinec@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

Evaluation exacte du signe d'un déterminant

Résumé : Ce document présente deux méthodes différentes pour calculer le signe d'un déterminant dont les entrées sont des entiers. La première de ces méthodes est basée sur le processus d'orthogonalisation de Gram-Schmidt et a été initialement proposée par Clarkson. Nous avons revisité l'analyse donnée par Clarkson et proposé une variante de sa méthode. La seconde méthode étend aux déterminants de dimension quelconque la méthode ABDPY proposée par Avnain et al. pour les déterminants de dimension deux et trois. Dans les deux cas, le signe d'un déterminant de dimension n formé d' entiers codés sur b-bits, est obtenu en utilisant une arithmétique exacte sur seulement $b + O(n)$-bits. De plus, les deux méthodes sont adaptatives c'est à dire d'autant plus rapide que la valeur du déterminant est éloignée de zéro.

Mots-clé : Géométrie algorithmique, arithmétique exacte, précision, algorithmes robustes

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 (for a discussion, see [C+96, 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 [SI89, SI94, Mil89, LM90, Hof89, HHK88, For92]). Such solutions, although very useful in some domains like solid modeling and CSG applications, are difficult 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 [FV93], Yap [Yap93, YD95], Burnikel and coll. [BKM+95], and many others. However, as reported for example by Karasick and coll. [KLN91], 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 [FV93] and has been shown experimentally to provide a substantial speed-up. Devillers and Preparata investigate the theoretical behavior of some filters [DP96]. In degenerate or near degenerate cases, however, exact arithmetic has to be carried out in full. Burnikel and coll. [BKM+95] and Yap [Yap93] 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 [Cla92] propose an efficient method to compute the sign of a determinant whose entries are integers. The so called ABDPY method, due to Avnaim and coll. [ABD+94], is an alternative solution for 2×2 and 3×3 determinants. Using an uncommon multiple-term arithmetic, Shewchuck [She96] designs an adaptive 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 revisit the method of Clarkson and extend the ABDPY method to higher dimensions.

Clarkson's method is based on the Gram-Schmidt orthogonalization process. We propose a variant of this algorithm, hereafter called the reorthogonalization method, which is somewhat simpler to analyze. This variant allows to compute the sign of an $n \times n$ deter-

minant whose entries are b-bit integers, that is, integers whose absolute values are smaller than 2^b, using an exact arithmetic on $b + \lceil 2.62(n-1) + 0.5 \log n \rceil$ bits, and with a worst case complexity of $O(bn^3 + n^3 \log n)$.

The lattice method extends the ABDPY method to higher dimensions. In fact, the lattice method borrows both from the ABDPY method and from Clarkson's method. Like the ABDPY method, it mostly consists in locating one of the column vectors of the determinant with respect to a region that approximates the hyperplane spanned by the others column vectors. When the endpoint of this 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 standard. Another factor of efficiency of those methods is their adaptability. 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

2.1 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 $A_1, \ldots, A_{k-1}$ (rather than $C_1, \ldots, C_{k-1}$ for standard Gram-Schmidt reductions). Thus, whatever the coefficients in this combinations are, $\det(\mathcal{A})$ is not affected as long as the linear combination is computed exactly.

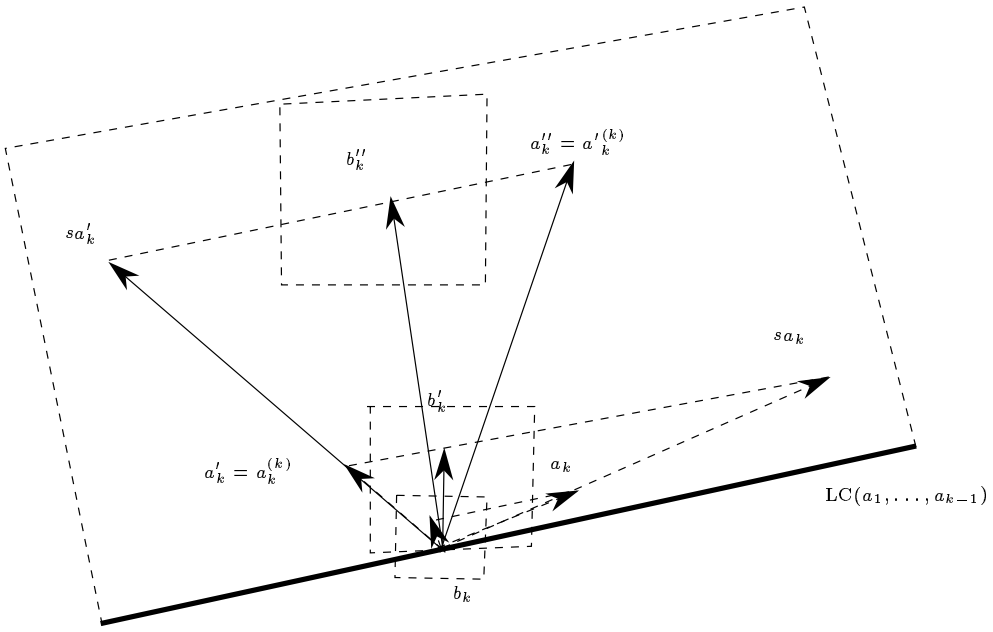

Figure 1: The process of reducing vector A_k is shown. At any time, A_k remains in the bounding box, but its component along the orthogonal of $(A_1, \ldots, A_{k-1})$ increases at each step. The computed vector B_k lies in the square box. At the first and second steps, the box does not ensure that the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

Furthermore, 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 roughly the same as that of [Cla92] (except that we use standard Gram-Schmidt reductions for the B_k's). Our variant is in the choice of s that leads to a much simpler analysis. 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.

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**

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, Clarkson shows that $\det(\mathcal{B}')$ is close to $\det(\mathcal{C}')$ and can be evaluated with relative precision less than 1 (using standard Gaussian elimination [Cla92, FM67]). Hence the sign of $\det(\mathcal{A})$ is computed exactly, and a good approximation $(\prod s)^{-1}\det(\mathcal{B}')$ of the value of $\det(\mathcal{A})$ 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 [Cla92], 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 on the precision $\mathbf{u}$ of the floating point computations.Assuming $\delta_n \leq 0.01$ implies only loose limits on n as a function of $\mathbf{u}$ that are satisfied for all practical values of n and $\mathbf{u}$.

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 preconditioning steps, and let S_k^c be defined as $\sum_{j=1}^{k-1} \|C_j\|^2$. The norms of all the vectors $A_k^{(j)}$ computed in line 11 satisfy

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

A similar statement is given in [Cla92], but we provide a short and self-contained proof of (1) in the appendix. It holds assuming that $\eta_k = 8\sum_{j=2}^{k} 2.06^{2(k-j)}\delta_j^2$ is smaller than 0.01. Since the δ_k's are very small, all the assumptions are satisfied for all practical values of n and $\mathbf{u}$. For instance, using doubles in the IEEE 754 standard, $\mathbf{u}$ is on the order of 2^{-53} and n must be less than 21.

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 [Cla92]) that $\left|S_k^b - S_k^c\right| \leq 0.07 S_k^c$.

Clarkson's choice [Cla92] 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)

$$s = \left\lceil fl\left(\sqrt{1.29 + \frac{S_k^b}{\lambda\left\|A'_k\right\|^2}}\right)\right\rfloor.$$

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 $S_k^b \leq \lambda\|A'_k\|^2$, and we know that $S_k^c \leq 1.07 S_k^b$. Plugging into (1) shows that $\left\|A_k^{(1)}\right\|^2 \leq 0.55(1+\lambda)\|A'_k\|^2$. We now set $\lambda = 0.45$ 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%.

If $s \geq 2$, we can say safely that $1.29 + \dfrac{S_k^b}{\lambda\|A'_k\|^2} \geq (s - 0.51)^2$, and hence

$$s^2\|A'_k\|^2 \leq \frac{s^2}{(s-0.51)^2 - 1.29}\frac{1.07}{\lambda}S_k^c \leq 10.23\, S_k^c, \tag{2}$$

since s is greater than 2 and λ is set at 0.45 . Plugging this bound into (1) for $j = 1$, we finally get that $\left\|A_k^{(1)}\right\|^2 \leq 6.14\, S_k^c$, which bounds the next vector A'_k.

From all this, we gather that $\|A'_k\|^2 \leq \max\left(\|A_k\|^2, 6.14 S_k^c\right)$ 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 $\left\|A_k^{(j)}\right\|^2 \leq \max\left(n2^{2b}, 6.14 S_k^c\right)$ 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 $S_k^c \leq n\, 6.14^{k-2} 2^{2b}$. Hence $b + \lceil 2.62(n-1) + 0.5\log n\rceil$ bits suffice to express all the vectors A_k and $A_k^{(j)}$ occurring in the algorithm.

2.2 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 other

iterations multiply the determinant by a value $s \geq 2$ while keeping the coefficients of $\mathcal{A}$ in the allowable range. Let t_2 be the number of iterations performed with $s \geq 2$. Since $2^{t_2}|\det(\mathcal{A})| \leq |\det(\mathcal{A}')| \leq n^{n/2}2^{bn}$, t_2 cannot exceed $(bn + n/2\log n)$ if $\det(\mathcal{A})$ is non zero. To bound the number t_1 of iterations with $s = 1$, note that

$$\left(\prod s\right)|\det(\mathcal{A})| = |\det(\mathcal{A}')| \leq \prod_{j=1}^{n} \|A'_j\| \leq 0.9^{t_1}\left(\prod s\right)\prod_{j=1}^{n}\|A_j\|.$$

Thus $|\det(\mathcal{A})| \leq 0.9^{t_1}n^{n/2}2^{bn}$ and t_1 cannot exceed $|\log 0.9|(bn + n/2\log n)$ if $\det(\mathcal{A})$ is non zero. Therefore, after at most $t_1 + t_2 = O(bn + n\log n)$ iterations, the algorithm is over or can stop concluding that $\det(\mathcal{A})$ 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 $P_k^c = \prod_{1\leq j\leq k}\|C_j\|$ is 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

3.1 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 the subspace $\mathbb{R}^{n-1}$ of $\mathbb{R}^n$ spanned by the first $(n-1)$ coordinate axis. Without loss of generality, $z_1, z_2, \ldots, z_n$ are assumed to be non negative. We choose the same point O for the origin of $\mathbb{R}^n$ and $\mathbb{R}^{n-1}$, and we note with the same capital letter U either a vector in $\mathbb{R}^n$ or the point $O + U$, and with the same small letter u either a vector in $\mathbb{R}^{n-1}$ or the point $O + u$.

Let H be the hyperplane passing through $\{O, U_1, U_2, \ldots, U_{n-1}\}$. The basic idea underlying the lattice method 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 $U_n = \mathrm{LC}(U_1, U_2, \ldots, U_{n-1}) + \alpha E_n$ where, as before LC denotes any linear combination of its arguments. Then, $D = \alpha \det(u_1, u_2, \ldots, u_{n-1})$ and the sign of α only depends on 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 circles. In fact, the lattice method consists in a first phase to locate the point U_n with respect to some region $\mathcal{H}$ of $\mathbb{R}^n$ which contains H and which 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 :

$$\mathcal{C} = u_1 \oplus u_2 \oplus \ldots \oplus u_{n-1} = \left\{ \sum_{i=1}^{n-1} \alpha_i u_i, \;\; 0 \leq \alpha_i < 1 \right\},$$

where the $\oplus$ symbol stands for a (half-closed) Minkowski sum. We note $\mathcal{C}(l_1, \ldots l_{n-1})$ the lattice cell that is the translated of $\mathcal{C}$ by the vector $\sum_{i=1}^{n-1} l_i u_i$. The point $c(l_1, \ldots l_{n-1}) = \sum_{i=1}^{n-1} l_i u_i$ is called the *reference point* of the cell $\mathcal{C}(l_1, \ldots l_{n-1})$. Analogous definitions and notations hold for the lattice $\mathcal{L}_H$ whose elementary cells partition H : the cell $\mathcal{C}_H(l_1, \ldots l_{n-1})$ of $\mathcal{L}_H$ is the copy of the origin cell $\mathcal{C}_H = U_1 \oplus U_2 \oplus \ldots \oplus U_{n-1}$ translated by the vector $C(l_1, \ldots l_{n-1}) = \sum_{i=1}^{n-1} l_i U_i$. The elementary cell $\mathcal{C}(l_1, \ldots l_{n-1})$ of $\mathcal{L}$ is the orthogonal projection of the cell $\mathcal{C}_H(l_1, \ldots l_{n-1})$ of $\mathcal{L}_H$.

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

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

Box $\mathcal{B}$ has the origin cell $\mathcal{C}$ of lattice $\mathcal{L}$ as orthogonal projection and contains the origin cell $\mathcal{C}_H$ of $\mathcal{L}_H$. Then, region $\mathcal{H}$ is defined as the union of all the boxes $\mathcal{B}(l_1, \ldots l_{n-1})$ which

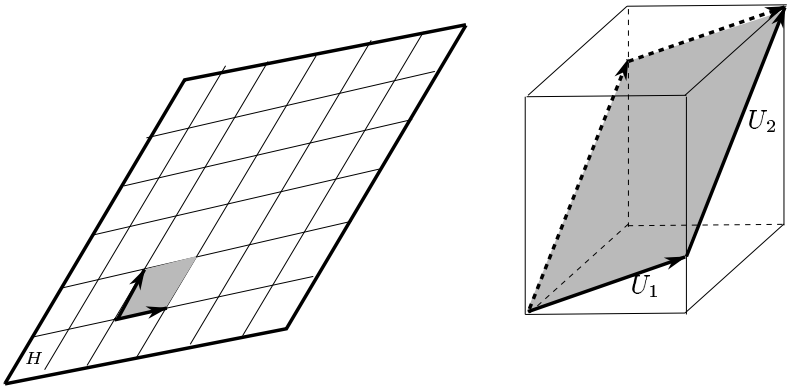

Figure 2: The cell $\mathcal{C}_H$ and the box $\mathcal{B}$

are copies of $\mathcal{B}$ translated by the vectors of lattice $\mathcal{L}_H$. Since $\mathcal{B}$ contains the origin cell $\mathcal{C}_H$, $\mathcal{B}(l_1, \ldots l_{n-1})$ contains cell $\mathcal{C}_H(l_1, \ldots l_{n-1})$ 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 evaluating the sign of a $(n-1) \times (n-1)$ determinant.
- Otherwise, U_n lies within some box $\mathcal{B}(k_1, \ldots k_{n-1})$ of $\mathcal{H}$ and the algorithm computes the vector $R = U_n - \sum_{i=1}^{n-1} k_i U_i$ which satisfies both following conditions $D = \det(U_1, U_2, \ldots, R)$ and $R \in \mathcal{B}$. Then, the algorithm enters the second phase.

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

$$D_t = \det(U_1, U_2, \ldots, R_t) = 2^t D \text{ and } R_t \in \mathcal{B}. \tag{3}$$

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

- If R' is found above or below $\mathcal{H}$ 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 $\mathcal{B}(k_1, \ldots k_{n-1})$ of $\mathcal{H}$. The algorithm computes the vector $R_{t+1} = R' - \sum_{i=1}^{n-1} k_i U_i$ which verifies $D_{t+1} = \det(U_1, U_2, \ldots, R_{t+1}) = 2^{t+1} D$ and $R_{t+1} \in \mathcal{B}$, and proceeds to the next iteration with D_{t+1} and R_{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 $bn + \left\lceil (\frac{n}{2} + 1) \log n \right\rceil$ 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 $2^t > n\sqrt{n}^n 2^{bn}$, which means that the number of performed iteration reaches $bn + \lceil (\frac{n}{2} + 1) \log n \rceil$, the algorithm can stop and conclude that D is null.

3.2 Further details.

In this subsection, we give the details of the differents operations performed during the two phases of the algorithm, in order to be able to bound the precision of the required arithmetic and the complexity of the whole algorithm in the next subsections.

The algorithm first computes the sign of the $(n-1)$ minors of D relatives to the nth row and discards the easy cases in which all those minors are null (in which case D is zero) or have alternate signs in which case the sign of D is known (recall that the components z_i are assumed to be positive). Thus at least one of those minors is non null and, permutating the vectors $\{U_1, U_2, \ldots, U_{n-1}\}$ if necessary, we may wlog assume that the minor $d_0 = \det(u_1, u_2, \ldots, u_{n-1})$ is non null.

First phase

The first phase of the algorithm is just like the location phase of the first iteration in the original ABDPY algorithm. To find out the position of U_n with respect to $\mathcal{H}$, the algorithm aims at finding the cell of $\mathcal{L}$ that contains the point u_n in order to compare the coordinate z_n with the z-range of the box of $\mathcal{H}$ projecting on this cell. To find the reference point c_u of the cell of $\mathcal{L}$ containing point u_n, the algorithm performs a dichotomic march in lattice $\mathcal{L}$ visiting a subset of the $(n-2)$-faces of lattice $\mathcal{L}$ that are intersected by the segment Ou_n. For each visited $(n-2)$-face f of $\mathcal{L}$, the z-range of the face f_H of $\mathcal{L}_H$ that projects on f is tested. If this range has no intersection with the z-range $[0, z_n]$ of points of segments OU_n, the relative position of U_n with respect to $\mathcal{H}$ is known and the march is stopped. If this case does not happen, the march yields the reference point c_u of the cell of $\mathcal{L}$ containing point u_n and the algorithm computes the vector $R = U_n - C_u$ where C_u is the point of H projecting on c_u.

More precisely, the following steps are performed

Step 1. The first step finds out which one of the 2^{n-1} cells of $\mathcal{L}$ incident to the origin is intersected by the segment Ou_n. This cell can be identified by computing the sign of the $n-1$ following $(n-1) \times (n-1)$-determinants

$$d_i = \det(u_1, \ldots, u_{i-1}, u_n, u_{i+1}, \ldots, u_{n-1}), \quad i = 1, \ldots, n-1.$$

It is always possible to find a permutation σ of $\{1, \ldots, n-1\}$ and $n-1$ values $\epsilon_i \in \{0, 1\}$ such that the vectors $v_i = (1 - 2\epsilon_{\sigma(i)})u_i$ verify :

$$\begin{aligned} d'_0 &= \det(v_1, v_2, \ldots, v_{n-1}) > 0 \\ d'_i &= \det(v_1, \ldots, v_{i-1}, u_n, v_{i+1}, \ldots, v_{n-1}) \geq 0, \quad i = 1, \ldots, n-1. \end{aligned}$$

If we consider now that the lattice $\mathcal{L}$ is generated by $\{v_1, v_2, \ldots, v_{n-1}\}$, the cell incident to the origin that is intersected by segment Ou_n is the elementary cell $\mathcal{C}' = v_1 \oplus v_2 \oplus \ldots \oplus v_{n-1}$ whose reference point is the origin. Denoting by c_u (resp. c_v) the reference point of the cell that contains u_n when we take $\{u_i\}$ (resp. $\{v_i\}$) as basis vectors of $\mathcal{L}$, we observe that

$$c_u = c_v - \sum \epsilon_i u_i. \tag{4}$$

We shall in fact compute c_v, and then obtain c_u using (4).

Step 2 Next, the algorithm has to find which facet of the origin cell $\mathcal{C}'$ is intersected by the ray originating from O in the direction of u_n. Here and in the following, we note

$$w = \sum_{i=1}^{n-1} v_i \tag{5}$$

the vertex of $\mathcal{C}'$ opposite to O, and

$$w_j = w - v_j, \quad j = 1, \ldots, n-1, \tag{6}$$

the vertices of $\mathcal{C}'$ adjacent to w. For $i = 1, \ldots, n-1$, we note h_i the $(n-2)$-hyperplane of $\mathbb{R}^{n-1}$ going through $\{O, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{n-1}\}$. The facets of $\mathcal{C}'$ incident to the origin are each included in one of the hyperplane h_i while the facets of $\mathcal{C}'$ incident to w are each included in one of the translated hyperplanes $h_i(1) = v_i + h_i$. Each $(n-3)$-face of $\mathcal{C}'$ incident to w (there are $(n-1)(n-2)/2$ such faces) is included in the affine hull $k_{i,j}$ of a set of points of the form $\{w, w - v_l \text{ with } l \in \{1, \ldots, n-1\},\ l \neq i, j\}$. Thus the facets of $\mathcal{C}'$ intersected by the ray supporting u is determined by the sign of the following $(n-1)\times(n-1)$-determinants :

$$e_{i,j} = \det(u_n, w, \underbrace{\ldots, w - v_l, \ldots}_{l \neq i,j}) = \det(u_n, w, , \underbrace{\ldots, -v_l, \ldots}_{l \neq i,j})$$

Each of those determinants allows to eliminate one of the $(n-1)$ facets of $\mathcal{C}'$ incident to w. Therefore at most $(n-2)$ determinants like $e_{i,j}$ have to be considered during Step 2 .

Step 3 In the following, we assume that the facet of $\mathcal{C}'$ intersected by the ray originating from O in the direction of u_n is the facet included in the hyperplane $h_1 + v_1$ and we note simply $h(m)$ the hyperplane $h_1 + mv_1$ translated from h_1 by the vector mv_1. (Note that there is no loss of generality in this assumption since it can always be achieved through a permutation of vectors $\{v_1, v_2, \ldots, v_{n-1}\}$.) In this step, we search the unique integer k such that $h(2^k)$ is intersected by Ou_n while $h(2^{k+1})$ is not (see Figure 3). In addition, for each $h(2^l)$ intersected by Ou_n the algorithm computes and stores in a stack (to be used in Step 4) the reference point $c(2^l)$ of the cell containing the intersection point $Ou_n \cap h(2^l)$. (Recall that cells are defined as semi-open sets including the facets incidents to their reference point but not the facets incident to the opposite vertex.) Furthermore, the algorithm compares the z-range $[0, z_n]$ of segment Ou_n with the z range of the face in lattice $\mathcal{L}_H$ that projects on the $(n-2)$-face of $\mathcal{L}$ containing the intersection point $Ou_n \cap h(2^l)$. If both ranges do not

overlap, the position of U_n with respect to $\mathcal{H}$ is known and the march is stopped. Otherwise the algorithm computes the z-coordinate of the point $C(2^l)$ of H projecting on $c(2^l)$ and probes the next hyperplane $h(2^{l+1})$. More precisely the following actions are performed.

Substep 3.1 This substep deals with hyperplane $h(1)$.

3.1.1 We decide if $h(1)$ is intersected by Ou_n by computing the sign of

$$f(1,1) = \det(u_n - v_1, v_2, \ldots, v_{n-1}) = \det(u_n - w_1, v_2, \ldots, v_{n-1}).$$

If h(1) is not intersected by u_n, then the march is over : u_n is included in cell $\mathcal{C}'$, whose reference point c_v is the origin O. Going to Step 6 the algorithm computes c_u using (4) and $R = U_n - C_u$ where C_u is the point of $\mathcal{L}_H$ projecting on c_u.

3.1.2 We set $c(1) = v_1$.

3.1.3 The $(n-2)$-face in $h(1)$ intersected by vector u_n is $c(1)+\mathcal{C}'_1$ where $\mathcal{C}'_1 = u_2 \oplus \ldots \oplus u_{n-1}$. We compute the z-coordinate of the point $C(1)$ of $\mathcal{L}_H$ projecting on $c(1)$ and the z-coordinates of the other vertices of the $(n-2)$-face of $\mathcal{L}_H$ projecting on $c(1) + \mathcal{C}'_1$. If the z-coordinates of those vertices are all negative or all greater than z_n, the position of U_n with respect to $\mathcal{H}$ is known and the march is dropped. Otherwise Substep 3.2 is entered.

Substep 3.2. In this substep, we successively probes hyperplanes $h(2), h(4) \ldots h(2^l) \ldots$. Assuming that, at a given stage, $h(2^l)$ has been found to intersect Ou_n and that points $c(2^l)$ and $C(2^l)$ have been computed, the following elementary actions are performed

3.2.1 We decide if Ou_n intersects hyperplane $h(2^{l+1})$ by computing the sign of

$$f(2^{l+1}, 1) = \det(u_n - 2c(2^l) - w_1, v_2, \ldots, v_{n-1}).$$

Notice that both $2c(2^l)$ and $2c(2^l) - w_1$ belong to hyperplane $h(2^{l+1})$, (see Figure 3). If u_n does not intersect $h(2^{l+1})$, Step 3 is over, go to Step 4.

3.2.2 Otherwise we compute $c(2^{l+1})$. Clearly $c(2^{l+1})$ is one of the 2^{n-2} vertices of the $(n-2)$-face $2c(2^l) + \mathcal{C}'_1$ (see Figure 4) and we can be decide which one by computing the sign of the $(n-1) \times (n-1)$ determinants

$$g(2^l, j) = \det(u_n, 2c(2^l) + w_1, \underbrace{\ldots, v_l, \ldots}_{l \neq 1, j}), \quad j = 2, \ldots, n-1.$$

3.2.3 The algorithm computes the points $C(2^{l+1})$ projecting on $c(2^{l+1})$ and test the z-range of the $(n-2)$-face in H projecting on $c(2^{l-1}) + \mathcal{C}'_1$. If the nth coordinates of the vertices of this face are all negative or all greater than z_n (recall that z_n is assumed to be non negative) the position of U_n with respect to H is known and the march is dropped. Otherwise the algotithm loops back to Substep 3.2.1 to test the next hyperplane .

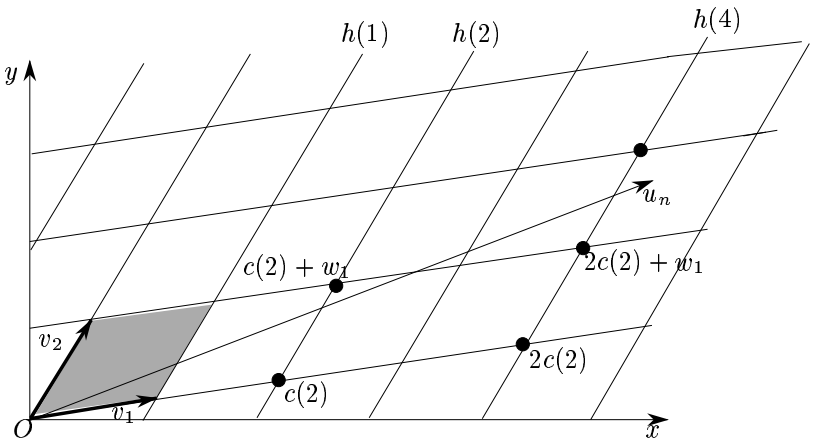

Figure 3: Localization of u_n

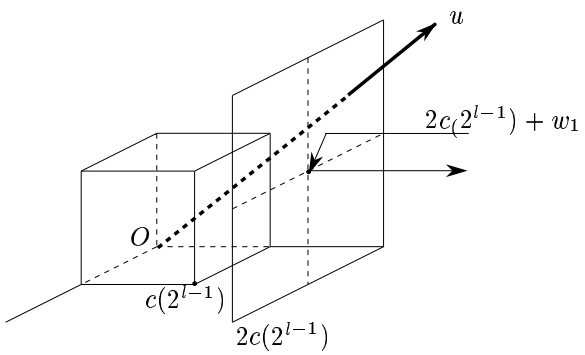

Figure 4: From $c(2^l)$ to $c(2^{l+1})$.

Step 4 Assume now that u_n intersects the hyperplane $h(2^k)$ but not the hyperplane $h(2^{k+1})$. Then the algorithm computes the integer m such that segment Ou_n intersects $h(m)$ but not $h(m+1)$. The determination of m is a binary search. This search involves $k-1$ stages numbered $k-1,\ldots,1$. At each stage l, the algorithm computes an integer m_l such that segment Ou_n intersects hyperplane $h(m_l)$ but not $h(m_l+2^l)$. The reference point $c(m_l)$ of the cell containing the intersection $h(m_l)\cap u_n$ is computed and the algorithm test the z-range of the $(n-2)$-face of $\mathcal{L}_H$ projecting on $c(m_l)+\mathcal{C}'_1$. More precisely, (assuming that $h(m_l)$ and $c(m_l)$ are known) the following actions are performed at stage $l-1$:

4.1 We decide whether Ou_n intersects $h(m_l+2^{l-1})$ or not by computing the sign of the determinant

$$f(m_l+2^{l-1},1)=\det(u_n-c(m_l)-c(2^{l-1})-w_1,v_2,\ldots,v_{n-1}),$$

where the vector $c(2^{l-1})$ is popped out from the stack. Notice that both $c(m_l)+c(2^{l-1})$ and $c(m_l)+c(2^{l-1})+w_1$ belong to $h(m_l+2^{l-1})$. If u_n does not intersect $h(m_l+2^{l-1})$ we have $m_{l-1}=m_l$ and $c(m_{l-1})=c(m_l)$ and we can pass to stage $l-2$. Otherwise, we have $m_{l-1}=m_l+2^{l-1}$ and we compute $c(m_{l-1})$ in the next substep.

4.2 Clearly $c(m_{l-1})$ is one of the 2^{n-2} vertices of the $(n-2)$-face $c(m_l)+c(2^{l-1})+\mathcal{C}'_1$ in $h(m_{l-1})$ and we can decide which one by computing the sign of the $(n-1)\times(n-1)$ determinants

$$g(m_l+2^{l-1},j)=\det(u_n,c(m_{l-1})+c(2^{l-1})+w_1,\underbrace{\ldots,v_i,\ldots}_{i\neq 1,j}),\quad j=2,\ldots,n-1.$$

4.3 The intersection $u_n\cap h(m_{l-1})$ lies in the $(n-2)$-face $c(m_{l-1})+\mathcal{C}'_1$. We compute the vertex $C(m_{l-1})$ of $\mathcal{L}_H$ projecting on $c(m_l)$ and test the z-range of the $(n-2)$ face of $\mathcal{L}_H$ projecting on $c(m_l)+\mathcal{C}'_1$. If the z-coordinates of the 2^{n-2} vertices of this face are all negative or all greater z_n, the position of U_n with respect to $\mathcal{H}$ is known and the march is dropped. Otherwise, go back to Substep 4.1 with $l-2$ or to Step 5 if $l-1=0$.

Step 5 At this point we know that vector Ou_n intersect $h(m_0)$ in the cell $c(m_0)+\mathcal{C}'_1$ but does not intersect $h(m_0+1)$. Thus the origin c_v of the cell containing u_n is one of the vertices of $c(m_0)+\mathcal{C}'_1$ and which one can be decided by computing the signs of the $(n-1)\times(n-1)$ determinants

$$f(m_0,j)=\det(u-c(m_0)-w_1,v_1,\underbrace{\ldots,v_l,\ldots}_{l\neq 1,j}),\quad j=2,\ldots,n-1.$$

Step 6 At last c_u is computed from c_v using 4 and vector $R=U_n-C_u$ is computed. By construction, the projection r of R belongs to the origin cell $\mathcal{C}$. If the z-coordinate of R is negative or greater than $\sum_{i=1}^{n-1}z_i$ the position of U_n with respect to H is known and the algorithm ends up. Otherwise, R belongs to box $\mathcal{B}$.

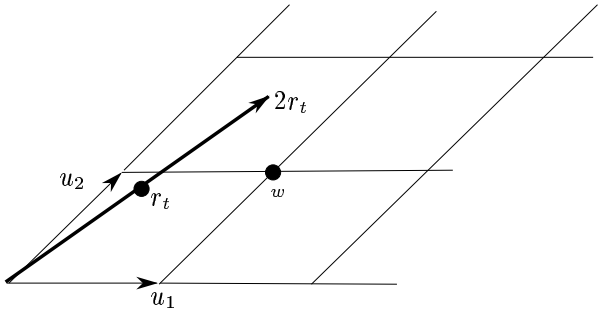

Figure 5: Locating $2r_t$ in lattice $\mathcal{L}$

Second 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 purpose, 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 cell $\mathcal{C}$ whose reference point is the origin, the reference point c_{t+1} is necessarily one of the 2^{n-1} vertices of $\mathcal{C}$ (see figure 5) and finding which one amounts to determine the sign of the following $(n-1) \times (n-1)$ determinants :

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

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 $R_{t+1} = 2R_t - C_{t+1}$, i. e.

$$r_{t+1} = 2r_t - c_{t+1} \tag{7}$$
$$z_{t+1} = 2z_t - z(C_{t+1}). \tag{8}$$

If z_{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 $R_{t+1} \in \mathcal{B}$, and the algorithm proceeds to iteration $t+1$.

3.3 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 here call nth level, the set of all the computations performed by the algorithm except those involved in the evaluations of signs of $(n-1) \times (n-1)$ determinants. More generally, the kth level is the set of all the computations performed by the algorithm to evaluate signs of

$k \times k$-determinants except those involved in the evaluations of signs 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. To be able to generalize our conclusions to lower levels, however, 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 $\sum_{i=1}^{n} \|U_i\|_\infty$ of the L_∞ norm of its column vectors.

Lemma 3.1 *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 all the computations performed at the nth level require an xact integer arithmetic on no more than $\lceil \log S_n \rceil + 1$ bits.*

To prove the lemma, we first consider in turn all the $(n-1)\times(n-1)$ determinants encountered during the two phases of the algorithm and prove for each of them that their input vectors can be computed exactly using no more than $\lceil \log S_n \rceil + 1$ bits and that their norm is less than $2S_n$. Then, we consider the other computations performed at the nth level.

The encountered $(n-1) \times (n-1)$ determinants. The claim is trivial for the determinants d_i encountered in Step 1 of the first phase. Then, we notice that the bound on the norm of the input determinant implies that the L_∞ norms of vectors w and w_j defined by equation 5 and 6 are bounded by S_n. Thus proposition of the lemma is also obviously true for the determinants and $e_{i,j}$ encountered in Step 2.

Let us consider the determinants $f(2^{l+1}, 1)$ encountered in Substeps 3.2.1. Those determinants involve, in addition to $(n-2)$ of the input vectors, a vector of the form $u_n - 2c(2^l) - w_1$. At the time this vector is considered, segment Ou_n is known to intersect hyperplane $h(2^l)$ on a point $p(2^l) = Ou_n \cap h(2^l)$ which belongs to the $(n-2)$-face $c(2^l) + \mathcal{C}_1^l$ in $h(2^l)$. Since $p(2^l)$ belongs to Ou_n, $\left\|p(2^l)\right\|_\infty$ and $\left\|u_n - p(2^l)\right\|_\infty$ are bounded by $\|U_n\|_\infty$. In addition, $\left\|p(2^l) - c(2^l)\right\|_\infty$ is bounded by $\sum_{i=1}^{n-1} \|U_i\|_\infty$. Thus, $\left\|u_n - c(2^l)\right\|_\infty$ and $\left\|c(2^l)\right\|_\infty$ are both bounded by S_n :

$$\begin{aligned} \left\|u - c(2^l)\right\|_\infty &\leq \left\|u_n - p(2^l)\right\|_\infty + \left\|p(2^l) - c(2^l)\right\|_\infty \leq S_n \\ \left\|c(2^l)\right\|_\infty &\leq \left\|p(2^l)\right\|_\infty + \left\|p(2^l) - c(2^l)\right\|_\infty \leq S_n, \end{aligned}$$

and the L_∞ norm of $\left\|u_n - 2c(2^l) - w_1\right\|_\infty$ does not exceed $3S_n$:

$$\left\|u_n - 2c(2^l) - w_1\right\|_\infty \leq \left\|u_n - c(2^l)\right\|_\infty + \left\|c(2^l)\right\|_\infty + \|w_1\|_\infty \leq 3S_n.$$

Now, if segment Ou_n intersects the hyperplane $h(2^{l+1})$, point $2c(2^l) + w_1$ is necesseraly one of the vertices of the $(n-2)$-face in that hyperplane intersected by Ou_n. Then the previous

argument shows that $\|u_n - 2c(2^l) - w_1\|_\infty$ does not exceed S_n. Therefore the algorithm can compute the components of $u_n - 2c(2^l) - w_1$ using the associative scheme :

$$u_n - 2c(2^l) - w_1 = [[u_n - c(2^l)] \ - \ c(2^l)] \ - \ w_1$$

If the components of this vector do not exceed S_n in absolute value, the above scheme do not require more than an exact integer arithmetic on $\lceil \log S_n \rceil + 1$ bits, and the norm of the $f(2^{l+1}, 1)$ is less than $2S_n$. Otherwise, we can conclude that Ou_n does not intersect $h(2^{l+1})$ and there is no use to test the determinant $f(2^{l+1}, 1)$ nor to compute the components of $u_n - 2c(2^l) - w_1$.

A similar argument apply to the determinant $f(1, 1)$ of Substep 3.1.1, to the determinants $f(m_l + 2^{l-1}, 1)$ encountered during Step 4.1 and to the determinants $f(m_0, j)$ encountered in Step 5.

Let us consider the determinants $g(2^{l+1}, j)$ encountered in Substep 3.2.2. Point $2c(2^l) + w_1$ is necessarily one of the vertices of the $(n-2)$-face in $h(2^{l+1})$ intersected by vector Ou_n which implies as above that its components do not exceed S_n in absolute value. Thus the norm of $g(2^{l+1}, j)$ is less than $2S_n$ and computing vector $2c(2^l) + w_1$ from $c(2^l)$ and w_1 does not require more than a $\lceil \log S_n \rceil + 1$ bits. A similar argument apply to the vectors $c(m_l) + c(2^{l-1}) + w_1$ and to the determinants $g(m_l + 2^{l-1}, j)$ encountered in Substep 4.2.

At last, let us consider the determinants a_i encountered during the second phase of the algorithm. Since r_t belongs to the origin cell $\mathcal{C}$ of $\mathcal{L}$, w is necessarily one of the vertices of the cell containing point $2r_t$ and the $\|2r^t - w\|_\infty$ is less than S_n. Thus vector $2r^t - w$ can be computed from r_t and w using no more than $\lceil \log S_n \rceil + 1$ bits and the norm of determinant a_i is less than $2S_n = S_{n-1}$.

The other computations performed at the nth level. These computations occurs in Substeps 3.1.3, 3.2.3 and 4.3 when computing the z-components of vertices $C(2^l)$ or $C(m_l)$, in Substep 6 when computing c_u from c_v and vector R and, at last, in each iteration of the second phase when computing C_{t+1} and R_{t+1}.

Let us first focus on Substep 3.2.3. We consider the 2^{n-2} vertices of $\mathcal{L}_H$ that project on the vertices of the $(n-2)$-face $c(2^l) + \mathcal{C}'_1$. As underlined in the description of Substep 3.2.3, if the z-coordinates of those vertices of $\mathcal{L}_H$ are all negative or all greater than z_n, the march is over and there is no need to compute $C(2^l)$. Otherwise, at least one of these vertices has its z-coordinate in the range $[0, z_n[$ which implies that the z-coordinate of any of them is less than S_n in absolute value. Thus, in particular the z-coordinate of $C(2^l)$ is less than S_n and can be computed exactly as the z-coordinate of $2C(2^{l-1}) + \sum_{i=2}^{n-1} U_i$ using no more than $\lceil \log S_n \rceil + 1$ bits. A similar argument apply to the computations performed in Substep 3.1.3 and 4.3.

In Step 6, c_v and c_u are both vertices of a cell of lattice $\mathcal{L}$ intersected by segment Ou_n. Thus $\|c_v\|_\infty$ and $\|c_v\|_\infty$ are less than S_n and computing c_u from c_v using 4 do not require more than $\lceil \log S_n \rceil + 1$ bits. Also, $\|u_n - c_u\|_\infty$ is bounded by S_n and vector $u - c_u$ can be computed on $\lceil \log S_n \rceil + 1$ bits. At last, as above, the z components of C_u and $R = U_n - C_U$ are not required unless both C_u and C_v have z components less than S_n in which case they can be computed exactly on $\lceil \log S_n \rceil + 1$ bits. Similar arguments apply for the computations

of $r_{t+1} = 2r_t - c_{t+1}$ and z_{t+1} at each iteration of the second phase. This ends the proof of Lemma 3.1. □

We are now in position to analyse the arithmetic precision required by the lattice method. An easy recurrence using the above lemma 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 [ABD+95], the whole algorithm requires only an exact integer arithmetic on $n - 2 + \lceil \log n \rceil + b$ bits.

3.4 Complexity.

Step 1, 2 and 5 involve altogether at most $3n-2$ evaluations of sign of $(n-1)\times(n-1)$ determinants. Now, the value m obtained in Step 4 is just the integer part of the coefficient of v_1 in the expression $u_n = LC(v_1, v_2, \ldots, v_{n-1})$. Therefore m is no more than $\frac{|\det(u_n, v_2, \ldots, v_{n-1})|}{|\det(v_1, v_2, \ldots, v_{n-1})|} \leq n\sqrt{n}^n 2^{bn}$ which means that the algorithm loops at most $bn + \lceil (\frac{n}{2} + 1) \log n \rceil$ times in Substep 3.2 and Step 4, computing in each loop $n-1$ signs of $(n-1) \times (n-1)$ determinants. 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. Thus the algorithms call for at most $O(bn^2 + n^2 \log n)$ evaluations of signs of $(n-1) \times (n-1)$ determinants. The other computations at the n-th level take time $O(n^2)$. Therefore, 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 implementation is at present time available up to dimension five only

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	149	604	2207	263	848	3354
Lattice	11	25	345	88	187	1232	352	635	2243	888	1491	4051
Filt.+Lat.	3	13	355	13	104	1171	13	288	2284	26	1060	4137
LN	3	53	53	4	295	296	na	na	na			

Table 1: 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

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 [FV93] for computing signs of determinants, 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 $b = 53 - (n - 2 + \lceil \log n \rceil)$. (This is the precision bound allowed for the lattice method and slightly over the bound allowed for reorthogonalization, but these bounds are pessimistic and, in practice, the reorthogonalization method 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 table 1.

The lattice and reorthogonalization methods show about the same performances, the balance being slightly in favor of the lattice method in dimension 3 and 4 and in favor of the reorthogonalization method in higher dimension. 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. For null determinants, both method are still faster than the exact computation of Leda and no more than 7 times slower than LN for 3×3 null determinants and 4 times slower for 4×4 null determinants. At least for dimension n up to 6, the timing results of the lattice and reorthogonalization methods agree with the predicted $O(n^3)$ law. This is confirmed by tests performed on the reorthogonalization method up to dimension 15

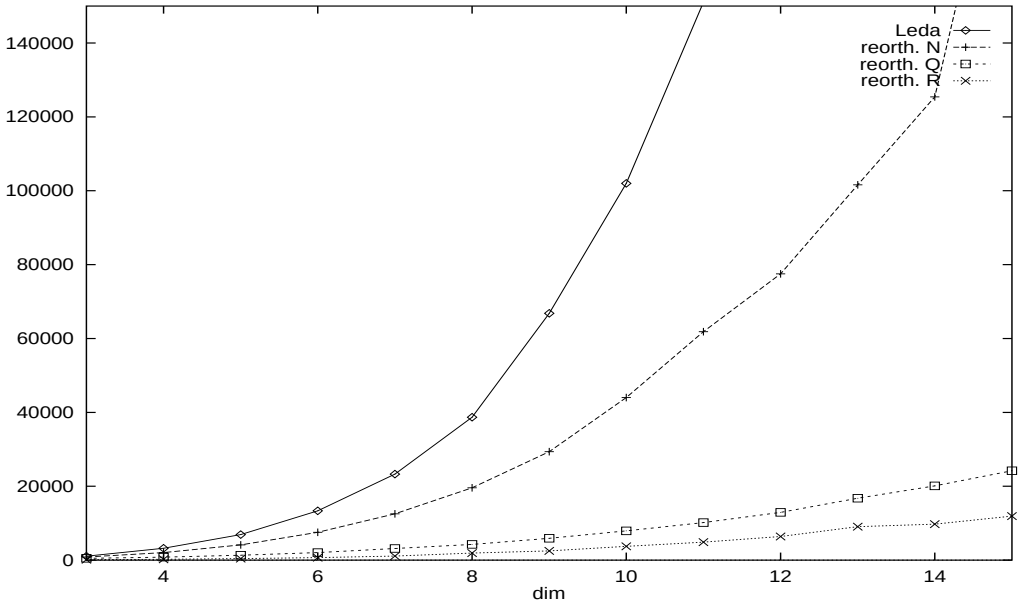

Figure 6: Timings as a function of the dimension, for the exact computation of signs of $n \times n$ determinants. The timings are in microsecond. The solid curve corresponds to an exact computation using a multi-precision package (here Leda). The three dotted curves correspond to the reorthogonalzation method on respectively null, quasi-null and random determinants.

and whose results are gathered in the graph of figure 6. Therefore it is clear that the comparison between the computation using a multi-precision package (Leda or the code generated by LN) and the lattice or reorthogonalization methods ends up sooner or later in favor of the latter methods when the dimension increases.

In fact, the lattice method could be very slow in some special cases, for instance when two column vectors of the input determinant are colinear. This agrees with the worst-case exponential behaviour, because in such cases most lower level determinants are null. However a simple preprocessing allows to avoid these bad cases. This preprocessing consists in first making sure that the first two column vectors are independant (by computing the 2×2 minors of these two columns) and then multiplying the input matrix both on the left and on the right side by a $n \times n$ matrix M whose entries are either 1 or -1 and whose determinant is not null. Such a preprocessing is very fast and only subtracts $\lceil \log^2 n \rceil$ bits from the allowed bit complexity of the entries.

Another caveat to be mentioned : any of the above method except Leda will fail in dimension n sufficiently large (possibly less than 20) due to overflow of the range of exponents in IEEE double precision. At last, we can also mention that these methods in dimension 6

have 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

The need for computing the sign of determinants arises in many geometric tests, such as orientation tests or in-sphere tests. In solid modeling, boundary representation require to compute the intersection of surfaces, which involves computing the sign of multivariate resultants, expressed as higher-dimensional determinants.

The reorthogonalization and the lattice method both appear to be practical methods to correctly compute the sign of determinants with integer entries, for dimension up to at least 15. The lattice method extends the ABDPY method (which is limited to the dimension 3) to any dimension. It requires fewer extra bits than the reorthogonalization method and offers equivalent performance. Furthermore, its special treatment of the last column makes it especially suitable for computing in-sphere tests. Both methods allow for reasonably big entries, are adaptive, and provide a dramatic increase of speed over standard multi-precision methods. This provides an affordable way of performing exact geometric tests, which is a decisive step towards robustness. However, some work still has to be done to provide good floating point filters for these high-dimensional geometric predicates.

References

[ABD+94] 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.

[ABD+95] 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.

[BKM+95] 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.

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

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.

[DP96] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. Rapport de recherche 2971, INRIA, 1996. also report CS96-27 Brown University.

[FM67] G. Forsythe and C. Moler. *Computer solutions of linear algebraic systems.* Prentice Hall, 1967.

[For92] 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.

[FV93] 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.

[HHK88] 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.

[Hof89] C. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[KLN91] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10:71–91, 1991.

[LM90] 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.

[Mil89] 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.

[She96] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[SI89] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inform. Proc.*, 12(4):380–393, 1989.

[SI94] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4:179–228, 1994.

[Yap93] C. K. Yap. Towards exact geometric computation. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 405–419, 1993.

[YD95] 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.

A Proof of (1).

Throughout the proof, we will assume that the unit roundoff $\mathbf{u}$ is small and that n is not too large. In [FM67], it is shown that a scalar product $X \cdot Y$ can be computed with error $1.01n\mathbf{u}\|x\|\|y\|$ and that a reduction factor $\left(\frac{X}{Y}\right)$ can be computed with error $1.01(2n+2)\mathbf{u}\frac{\|X\|}{\|Y\|}$, as long as $n\mathbf{u} \leq 0.01$. For all purposes, we will assume that $3n\mathbf{u} \leq 0.01$ and that $\delta_j \leq 0.01$.

If $j < l \leq k$, we know that $A_k^{(j)} = A_k^{(l)} + \mathrm{LC}(A_j, \ldots, A_{l-1})$. But the vector C_l is orthogonal by construction to $A_j, \ldots, A_{l-1}$ and hence $A_k^{(j)}$ and $A_k^{(l)}$ have the same component along C_l. Moreover $A_k^{(k)}$ has component sC_k along C_k. Thus for any $j \leq k$, decomposing $A_k^{(j)}$ on the orthogonal basis $C_1, \ldots, C_k$ yields

$$A_k^{(j)} = sC_k + \sum_{l=j}^{k-1} \left(\frac{A_k^{(l)}}{C_l}\right) C_l.$$

We reduce A_k exactly only when the preconditioning invariant is not satisfied, which implies that $\|C_k\|^2 \leq 0.53\|A_k'\|^2$, accounting for the roundoff errors in checking that condition. This bounds the first component of $\left\|A_k^{(j)}\right\|$. Using the following bound (proven below) on the other components in the orthogonal basis $C_1, \ldots, C_{k-1}$:

$$\left(\frac{A_k^{(j)}}{C_l}\right)^2 \|C_l\|^2 \quad \leq \quad 2\delta_{l+1}^2 \left\|A_k^{(j+1)}\right\|^2 + 0.5\|C_l\|^2, \tag{9}$$

we find that

$$\left\|A_k^{(j)}\right\|^2 \leq 0.53\, s^2\|A_k'\|^2 + 0.5 S_k^c + \sum_{l=j+1}^{k} 2\delta_l^2 \left\|A_k^{(l)}\right\|^2. \tag{10}$$

We finish the proof of the lemma by using the bound (proven below)

$$\left\|A_k^{(l)}\right\| \leq 2.06^{k-l} s\|A_k'\| + 0.71 \sum_{j=l}^{k-1} 2.06^{j-l}\|C_j\|. \tag{11}$$

Using the Cauchy-Schwartz inequality, we get

$$\left(\sum_{j=l}^{k-1} 2.06^{j-l}\|C_j\|\right)^2 \quad \leq \quad \left(\sum_{j=l}^{k-1} 2.06^{2(j-l)}\right)\left(\sum_{j=l}^{k-1}\|C_j\|^2\right) \leq \frac{2.06^{2(k-l)}}{2.06^2 - 1} S_k^c.$$

Note that $0.71/(2.06^2-1) \leq 1$. Hence, squaring (11) and using the classical inequality $(x+y)^2 \leq 2(x^2+y^2)$, we finally obtain

$$\left\| A_k^{(l)} \right\|^2 \leq 2 \times 2.06^{2(k-l)} \left(s^2 \|A'_k\|^2 + S_k^c \right).$$

Using this last bound in (10) yields

$$\left\| A_k^{(j)} \right\|^2 \quad \leq \quad 0.53(1+\eta_k)\, s^2 \|A'_k\|^2 + 0.5(1+1.01\eta_k) S_k^c$$

where $\eta_k = 8 \sum_{j=2}^{k} 2.06^{2(k-j)} \delta_j^2$. Finally, inequality (1) is implied by the assumption $\eta_k \leq 0.01$. The proof of (1) is thus complete, save for (9) and (11).

To prove (11), first observe that because $A_j - C_j$ and C_j are orthogonal, and because of the preconditioning invariant, we have

$$\|A_j - C_j\|^2 = {A_j}^2 - {C_j}^2 \leq (2.05 - (1-\delta_j)^2) {B_j}^2 \leq 1.07 \|B_j\|^2,$$

assuming that $\delta_j \leq 0.01$. But then

$$\begin{aligned}
\left\| A_k^{(j+1)} - \left(\frac{A_k^{(j+1)}}{B_j} \right) A_j \right\| \quad &\leq \quad \left\| A_k^{(j+1)} - \left(\frac{A_k^{(j+1)}}{C_j} \right) C_j \right\| + \left| \frac{A_k^{(j+1)}}{C_j} - \frac{A_k^{(j+1)}}{B_j} \right| \|C_j\| + \\
&\qquad\qquad \left| \frac{A_k^{(j+1)}}{B_j} \right| \|A_j - C_j\| \\
&\leq \quad \left\| A_k^{(j+1)} \right\| + \left\| A_k^{(j+1)} \right\| \frac{\|B_j - C_j\|}{\|B_j\|} + \sqrt{1.07} \left\| A_k^{(j+1)} \right\| \\
&\leq \quad (1 + \delta_j + \sqrt{1.07}) \left\| A_k^{(j+1)} \right\| \leq 2.05 \left\| A_k^{(j+1)} \right\|.
\end{aligned}$$

However, by the definition of $A_k^{(j)}$,

$$\left\| A_k^{(j)} \right\| \quad \leq \quad \left\| A_k^{(j+1)} - \left(\frac{A_k^{(j+1)}}{B_j} \right) A_j \right\| + \left| \frac{A_k^{(j+1)}}{B_j} - fl\left(\frac{A_k^{(j+1)}}{B_j} \right) \right| \|A_j\| + \frac{1}{2} \|A_j\|. \quad (12)$$

The first term is bounded by $2.05 \left\| A_k^{(j+1)} \right\|$ and the third by $0.5\|A_j\|$. The second involves the error on the floating point computation of the reduction factor and can be bounded by $0.01 \left\| A_k^{(j+1)} \right\|$ for all practical values of n and $\mathbf{u}$. Hence $\left\| A_k^{(j)} \right\|$ is bounded above by $2.06 \left\| A_k^{(j+1)} \right\| + 0.5\|A_j\|$. The preconditioning invariant implies that $\|A_j\| \leq \sqrt{2.01}\|C_j\|$. Thus,

$$\left\| A_k^{(j)} \right\| \leq 2.06 \left\| A_k^{(j+1)} \right\| + 0.5\|A_j\| \leq 2.06 \left\| A_k^{(j+1)} \right\| + 0.71\|C_j\|.$$

Unrolling this recurrence equation with $A_k^{(k)} = sA'_k$, we obtain (11).

To prove (9), remember that $\left(\frac{A_j}{C_j}\right) = 1$. Reducing (12) by C_j yields

$$\left|\frac{A_k^{(j)}}{C_j}\right|\|C_j\| \quad \leq \quad \left|\frac{A_k^{(j+1)}}{C_j} - \frac{A_k^{(j+1)}}{B_j}\right|\|C_j\| + \left|\frac{A_k^{(j+1)}}{B_j} - fl\left(\frac{A_k^{(j+1)}}{B_j}\right)\right|\|C_j\| + \frac{1}{2}\|C_j\|$$

The first term is less than $\delta_j \left\| A_k^{(j+1)} \right\|$, and the third is less than $0.5\|C_j\|$. The second involves only the error on the floating point computation of the reduction factor and combined to the first term can be bounded by $\delta_{j+1} \left\| A_k^{(j+1)} \right\|$. Hence the norm of the component of $A_k^{(j+1)}$ along C_j is bounded by $\delta_{j+1} \left\| A_k^{(j+1)} \right\| + 0.5\|C_j\|$. Again, one obtains (9) by squaring and using the classical inequality $(x+y)^2 \leq 2(x^2+y^2)$. □

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399