Computing Exact Geometric Predicates Using Modular Arithmetic with Single Precision

Hervé Brönnimann, Ioannis Z. Emiris, Victor Y. Y. Pan, Sylvain Pion

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Computing exact geometric predicates using modular arithmetic with single precision

Hervé Brönnimann* , Ioannis Z. Emiris* , Victor Y. Pan‡ , Sylvain Pion*

N˚ 3213

July 1997

THÈME 2

ISSN 0249-6399

Computing exact geometric predicates using modular arithmetic with single precision

Hervé Brönnimann* , Ioannis Z. Emiris* , Victor Y. Pan‡ , Sylvain Pion*

Thème 2 — Génie logiciel
et calcul symbolique
Projet Prisme

Rapport de recherche n ° 3213 — July 1997 — 29 pages

Abstract: We propose an efficient method that determines the sign of a multivariate polynomial expression with integer coefficients. This is a central operation on which the robustness of many geometric algorithms depends. Our method relies on modular computations, for which comparisons are usually thought to require multiprecision. Our novel technique of *recursive relaxation of the moduli* enables us to carry out sign determination and comparisons by using only floating point computations in single precision. This leads us to propose a hybrid symbolic-numeric approach to exact arithmetic. The method is highly parallelizable and is the fastest of all known multiprecision methods from a complexity point of view. As an application, we show how to compute a few geometric predicates that reduce to matrix determinants and we discuss implementation efficiency, which can be enhanced by arithmetic filters. We substantiate these claims by experimental results and comparisons to other existing approaches. Our method can be used to generate robust and efficient implementations of geometric algorithms (convex hulls, Delaunay triangulations, arrangements) and numerical computer algebra (algebraic representation of curves and points, symbolic perturbation, Sturm sequences and multivariate resultants).

Key-words: Computational geometry, exact arithmetic, robustness, modular computations, single precision, (RNS) Residue Number Systems

(Résumé : tsvp)

A preliminary version appeared in [7]. This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL). Work on this paper by Victor Y. Pan was performed while visiting the second author and was supported by NSF Grants CCR 9020690 and 9625344 and PS-CUNY Awards 666327 and 667340.

* INRIA Sophia-Antipolis, B.P. 93, 2004, Route des Lucioles, 06902 Sophia-Antipolis Cedex, FRANCE.
‡ Department of Mathematics and Computer Science, Lehman College, City University of New-York, Bronx NY 10468, USA.

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

Calcul exact de prédicats géométriques par une arithmétique modulaire en simple précision

Résumé : Nous proposons une technique efficace pour retrouver le signe d'une expression polynomiale à coefficients entiers. Cette opération est essentielle pour la robustesse des algorithmes géométriques. Notre méthode repose sur l'arithmétique modulaire, une arithmétique efficace mais pour laquelle il est généralement admis que les comparaisons requièrent une précision multiple. Notre algorithme d'*élimination récursive des modulis* nous permet de déterminer le signe et d'effectuer des comparaisons dans ce modèle en simple précision. Cette méthode est hautement parallélisable. Nous montrons par exemple comment calculer certains prédicats géométriques qui se réduisent à calculer le signe d'un déterminant. Nous discutons les détails d'implantation, et en particulier l'utilisation de filtres arithmétiques. Nos méthodes peuvent être utilisées pour implanter de nombreux algorithmes géométriques (enveloppe convexe, triangulations de Delaunay, arrangements), ainsi que d'algèbre numérique (représentation de nombres, courbes et surfaces algébriques, perturbation symbolique, théorie de Sturm et résultants multivariés).

Mots-clé : Géométrie algorithmique, arithmétique exacte, robustesse, calcul modulaire, simple précision

1 Introduction

In computational geometry and computer graphics, floating point (f.p.) arithmetic is extremely popular because of its speed. Most of geometric predicates can be expressed as computing the sign of an algebraic expression, which can be achieved by using f.p. arithmetic with a fixed finite precision. Unfortunately, the roundoff errors may easily lead to the wrong sign, causing the algorithm to fail on the input. This problem is often referred to as the *robustness* problem [25]. One solution to the robustness problem is to explicitly handle numerical inaccuracies, so as to design an algorithm that does not fail even if the numerical part of the computation is done approximately [27, 39], or to analyze the error due to the f.p. imprecision [19]. Such designs are extremely involved and have only been done for a few algorithms. The general solution, it has been widely argued, is to compute the predicates exactly [9, 16, 18, 20, 44]. (See also section 6.2.) This is also the position taken by this paper. This goal can be achieved in many ways: computing the algebraic expressions with infinite precision [42], with a finite but much higher precision that can be shown to suffice [21], or by using an algorithm that performs a specific test exactly. In the last category, much work has focused on computing the sign of the determinant of a matrix with integer entries [3, 6, 12], which applies to many geometric tests (such as orientation tests, in-circle tests, comparing segment intersections) as well as to algebraic primitives (such as resultants and algebraic representations of curves and surfaces). Recently, some techniques have been devised for handling arbitrary polynomial expressions and f.p. representation [38] but their complexity grows rather fast with that of the computation.

In computer algebra and symbolic computation, on the other hand, exact arithmetic is almost always assumed. When approximate calculation is not an option, a popular approach is to use big-integer and big-float multiprecision packages. This implies that operands are computed and stored with arbitrary precision, including intermediate quantities whose magnitude may be significantly larger than that of the output values. To remedy this problem, a substantial amount of work in the area has focused on modular arithmetic, which allows most of the computation to be carried over fixed precision integers. However, the modular representation of a rational number is typically not sufficient, and most problems require the reconstruction of the exact number, which means that some arbitrary precision is still required. Real algebraic numbers are represented as the unique root of a given polynomial in a given interval. Such a representation can be computed by applying Sturm theory. Besides the computation of Sturm sequences, finding the isolating interval requires many computations of signs of polynomial expressions with integer coefficients. One major drawback of these methods is the slowdown due to the handling of full precision.

In this paper, we propose a method that determines the sign of a multivariate polynomial expression with integer coefficients, using no operations other than modular arithmetic and f.p. computations with a fixed finite (single) precision, thus removing the need for arbitrary precision computations. These operations can be performed very fast on usual computers (see section 2). The Chinese remainder theorem enables us to perform rational algebraic computations modulo several primes, that is, with a lower precision, and then to combine them together in order to recover the desired output value. The latter stage of combining the

values modulo smaller primes, however, was always considered a bottleneck of this approach, because higher precision computations were required at this stage. Our paper proposes a new technique, which we call *recursive relaxation of the moduli* and which enables us to resolve the latter problem (section 3). Thanks to this technique, we correctly recover the sign of an integer from its value reduced modulo several smaller primes, and we only use some simple lower precision computations at the recovery stage. The worst-case complexities of these algorithms are no worse than quadratic in the number of finite fields, which is the same as for the standard multiprecision packages. The best worst-case complexities can in theory be lowered to quasi-linear in the asymptotic notation, albeit with a huge overhead. In practice, our algorithms are expected to behave linearly except in some cases whose probabilities are extremely small, and they entail little or no overhead. We therefore believe them to be of extremely practical value. In section 6, we then show how to compute a few geometric predicates that reduce to computing signs of matrix determinants. Preliminary experimental results and running times are discussed in section 7. In general, our methods are comparable in speed to other exact methods, and even faster for particular inputs.

Related work. Performing exact arithmetic is usually expensive. Thus, it is customary to resort to arithmetic filters [21]: those filters safely evaluate a predicate in most cases, in order to avoid performing a more expensive exact implementation. The difficult cases arise when the expression whose sign we wish to compute is very small. For typical filters, the smaller this quantity, the slower the filter [12, 3, 38]: this is referred to as *adaptivity*. Modular arithmetic displays an opposite kind of adaptivity: with a smaller quantity, fewer moduli have to be computed, hence the test is faster. Typically, when filters fail, they also provide an upper bound on the absolute value of the expression whose sign we wish to compute (see many details and estimates in [35, 15]). This bound can then be used to determine how many moduli should be taken. Modular arithmetic is therefore complementary to the filtering approach. We also observe this in section 7.

Residue Number Systems (RNS) express and manipulate integers of arbitrary precision by their residues with respect to a given set of numbers, the moduli. They are popular because they provide a cheap and highly parallelizable version of multiprecision arithmetic. It is impossible here to give a fair and full account on RNS, but Knuth [30] and Aho, Hopcroft, and Ullman [1] provide a good introduction to the topic. From a complexity point of view, RNS allows to add and multiply numbers in linear time. Their weak point is that sign computation and comparisons are not easily performed and seem to require full reconstruction in multiple precision, which defeats its purpose. This is precisely the issue that our paper handles.

The closest predecessors of our work are apparently [17] and [28]. The algorithm of Hung and Parhami [28] corresponds to single application of the second stage of our recursive relaxation of the moduli. Such a single application suffices in the context of the goal of [28], that is, application to divisions in RNS, but in terms of the sign determination of an integer, this only works for an absolutely larger input. The paper [17] gives probabilistic estimates for early termination of Newton's interpolation process, which we apply in our

probabilistic analysis of our algorithm 4. Its main subject is an implementation of an algorithm computing multidimensional convex hulls. The paper [17] does not use our techniques of recursive relaxation of the moduli, and it does not contain the basic equations (1)–(3) of our section 3.1.

2 Exact sign computation using modular arithmetic

Floating point (f.p.) computations. Our model of a computer is that of a f.p. processor that performs operations at unit cost by using b-bit precision (e.g., in the IEEE 754 double precision standard, we have $b = 53$). It is a realistic model as it covers the case of most workstations used in research and industry [23, 30, 38]. We will use mainly one basic property of f.p. arithmetic on such a computer: for all four arithmetic operations, the computed result is always the f.p. representation that best approximates the exact result. This means that the relative error incurred by an operation returning x is at most 2^{-b-1}, and that the absolute error[1] is at most $2^{\lfloor \log |x| - b - 1 \rfloor}$. In particular, operations performed on pairs of integers smaller than 2^b are performed exactly as long as the result is also smaller than 2^b.

To be able to discuss the properties of f.p. arithmetic, it is convenient to introduce the following notation [38]: given any real number x, it is *representable*[2] over b bits if $x = 0$ or if $x2^{-\lfloor \log x \rfloor + b}$ is an integer; $\widetilde{x}$ denotes the representable f.p. number closest to x (with any tie-breaking rule if x is right in-between two representable numbers), and $\mathrm{ulp}(x)$ denotes the *unit in the last place*, that is, $2^{\lfloor \log |x| - b \rfloor}$ if $x \neq 0$, and 0 otherwise. With this notation, the absolute error in computing an operation that returns x is $\frac{1}{2}\mathrm{ulp}(x)$.

Modular computations. Let $m_1, \ldots, m_k$ be k pairwise relatively prime integers and let $m = \prod_i m_i$. For any number x (not necessarily an integer), we let $x_i = x \bmod m_i$ be the only number in the range $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ such that $x_i - x$ is a multiple of m_i. (This operation is always among the standard operations because it is needed for reducing the arguments of periodic functions.)

This operation can be extended modulo an f.p. number as follows: an f.p. number x is truncated to a non-null f.p. number y and the result is defined as $x - \lceil x/y \rfloor y$. Therefore, $x \bmod m_i$ is the result of truncating x to m_i, and the (signed) fractional part $\mathrm{frac}(x)$ of x is the result of truncating x to 1. Note that the result of truncating x to a power of two is always representable if x is representable.

To be able to perform arithmetic modulo m_i on integers by using f.p. arithmetic with b-bit precision, we will assume that $m_i \leq 2^{b/2+1}$. Performing modular multiplications of two integers from the interval $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ can be done by multiplying these numbers and returning their product modulo m_i. (The product is smaller than 2^b in magnitude and hence is computed exactly.) Performing additions can be done very much in the same way, but since

[1] All logarithms in this paper are base 2.

[2] We systematically ignore underflows and overflows, by assuming that the range of exponent is large enough. A few modern packages now provide f.p. arithmetic with the exponent stored in a separate integers, which extends the IEEE 754 double precision standard by quite a lot.

the result is in the range $[-m_i, m_i)$, taking the sum modulo m_i can be achieved by adding or subtracting m_i if necessary. Modular divisions can be computed using the extended Euclid's algorithm; we will need them in this paper only in section 6. Therefore, arithmetic modulo m_i can be performed on integers by using f.p. arithmetic with b-bit precision, provided that $m_i \leq 2^{b/2+1}$.

Exact sign computation. In this paper, we consider the following computational problem.

Problem 1 *Let k, b, $m_1, \ldots, m_k$ denote positive integers, $m_1, \ldots, m_k$ being pairwise relatively prime, such that $m_i \leq 2^{b/2+1}$, and let $m = \prod_{i=1}^{k} m_i$. Let x be an integer whose magnitude is smaller than $\lfloor (m/2)(1 - 3k2^{-b-1}) \rfloor$. Given $x_i = x \bmod m_i$, compute the sign of x by using only modular and floating-point arithmetic both performed with b-bit precision.*

We will solve this problem, even though x can be huge and, therefore, not even representable by using b bits. In the worst case, our solutions require $O(k^2)$ operations and therefore do not improve asymptotically over the standard multiprecision approach. They are simple, however, and require little or no overhead. In practice, they only perform $O(k)$ operations. Thus they are very well suited for implementation.

3 Lagrange's method

According to the Chinese remainder theorem, x is uniquely determined by its residues x_i, that is, Problem 1 is well defined and admits a unique solution. Moreover, this solution can be derived algorithmically from a formula due to Lagrange. A comprehensive account of this approach can be found in [30, 31].

3.1 The basic method

This section describes the basic algorithm relying on Lagrange's approach. If x is an integer in the range $\left[-\frac{m}{2}, \frac{m}{2}\right)$, x_i stands for the residue $x \bmod m_i$, $v_i = m/m_i = \prod_{j \neq i} m_j$, and $w_i = v_i^{-1} \bmod m_i$, then

$$x = \left(\sum_{i=1}^{k} ((x_i w_i) \bmod m_i)\, v_i \right) \bmod m. \tag{1}$$

Trying to determine the sign of such an integer, we compute the latter sum approximately in fixed b-bit precision. Computing a linear combination of large integers v_i with its subsequent reduction modulo m can be difficult, so we prefer to compute the number

$$S = \frac{x}{m} = \operatorname{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i} \right),$$

where $\mathrm{frac}(z)$ is the fractional part of a number z that belongs to $\left[-\frac{1}{2}, \frac{1}{2}\right)$.

If S were computed exactly, then we would have $S = x/m$, due to Lagrange's interpolation formula. In fact, S is computed with a fixed b-bit precision. Nevertheless, if we compute it by incrementally adding the ith term and taking fractional part, the error follows the induction

$$\varepsilon_i = \varepsilon_{i-1} + 2^{-b-1} + 2^{-b},$$

where the term 2^{-b-1} accounts for the error on computing the ith term of S, and the term 2^{-b} accounts for the error on computing the incremental sum. Moreover, $\varepsilon_1 = 2^{-b-1}$. Hence S approximates x/m with an absolute error $\varepsilon_k = (3k-2)2^{-b-1}$. Therefore, if $|S|$ is greater than ε_k, the sign of x is the same as the sign of S, and we are done. Otherwise, $|x| < 2\varepsilon_k m$. Since $m_k \leq 2^{b/2+1}$, we can say conservatively that for all practical values of k and b, this is smaller than $\frac{m}{2m_k}(1-\varepsilon_{k-1})$, and hence we may recover x already from $x_i = x \bmod m_i$ for $i = 1, \ldots, k-1$, that is, it suffices to repeat the computation using only $k-1$, rather than k moduli. Recursively, we will reduce the solution to the case of a single modulus m_1 where $x = x_1$. We will call this technique *recursive relaxation of the moduli*, and we will also apply it in section 3.2.

We will present our resulting algorithm by using additional notation:

$$\begin{aligned}
m^{(j)} &= \prod_{1 \leq i \leq j} m_i, \\
v_i^{(j)} &= \prod_{\substack{1 \leq \ell \leq j \\ \ell \neq i}} m_\ell, \\
w_i^{(j)} &= \left(v_i^{(j)}\right)^{-1} \bmod m_i, \\
S^{(j)} &= \mathrm{frac}\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i}\right),
\end{aligned}$$

so that $m = m^{(k)}$, $v_i = v_i^{(k)}$, $w_i = w_i^{(k)}$ and $S = S^{(k)}$. All the computations in this algorithm are performed by using f.p. arithmetic with b-bit precision.

Note that $w_i^{(j)} = w_i^{(j+1)} m_{j+1} \bmod m_i$, hence the i-th term in $S^{(j)}$ can be computed from the i-th term in $S^{(j+1)}$. Thus only the $w_i^{(k)}$'s are needed in the algorithm. Since k is specified in the input, though, a quadratic table still seems to be necessary. This can be avoided without substantial slowdown (see the remark below).

Algorithm 1 : Compute the sign of x knowing $x_i = x \bmod m_i$ for all $1 \leq i \leq k$

Precomputed data: *m_j, $w_i^{(k)}$, ε_j, for all $1 \leq i \leq k$*
Input: *integers k and $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, for all $1 \leq i \leq k$*
Output: *sign of x, the unique solution of $x_i = x \bmod m_i$ in $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$*
Precondition: $|x| \leq \frac{m^{(k)}}{2}(1-\varepsilon_k)$

1. *Let* $j \leftarrow k+1$, $z_i = x_i w_i^{(k)} \bmod m_i$ *for all* $1 \le i \le k$
2. *Repeat* $j \leftarrow j-1$,
   $S^{(j)} \leftarrow frac\left(\sum_{i=1}^{j} \frac{z_i}{m_i}\right)$
   if $|S^{(j)}| < \varepsilon_j$ *and* $j > 0$ *then* $z_i \leftarrow z_i m_j \bmod m_i$ *for all* $1 \le i < j$
   until $|S^{(j)}| > \varepsilon_j$ *or* $j = 0$
3. *If* $j = 0$ *return* "$x = 0$"
4. *If* $S^{(j)} > 0$ *return* "$x > 0$"
5. *If* $S^{(j)} < 0$ *return* "$x < 0$"

The following lemma bounds the number of operations performed by the algorithm in the worst case.

Lemma 3.1 *Algorithm 1 computes the sign of x knowing its residues x_i by using at most $\frac{k(k+1)}{2}$ modular multiplications, $\frac{k(k+1)}{2}$ f.p. divisions, $\frac{k(k+1)}{2}$ f.p. additions, and $k+2$ f.p. comparisons.*

Proof. The m_i's and the $w_i^{(j)}$'s are computed once and for all and placed into a table, so they are assumed to be available to the algorithm at unit cost. In step 2, a total of j modular multiplications, j f.p. divisions, j f.p. additions (including taking the fractional part) and one comparison are performed. □

In almost all practical instances of the problem, $|x|$ is on the same order of magnitude as $m^{(k)}$. If $|x|$ is not too small compared to $m^{(k)}$, then only step k is performed, involving only at most k f.p. operations of each kind. This is to be contrasted with full reconstruction, which requires $\Theta(k^2)$ operations. Thus algorithm 1 is of great practical value.

By using parallel implementation of the summation of k numbers on $\lceil k/\log k \rceil$ arithmetic processors in $2\lceil \log k \rceil$ time (cf. e.g. [5, ch.4]), we may perform algorithm 1 on $\lceil k/\log k \rceil$ arithmetic processors in $O(k \log k)$ time, assuming each b-bit f.p. operation takes constant time. Furthermore, if $\lceil k^2/\log k \rceil$ processors are available, we may compute all the $S^{(j)}$ and compare $|S^{(j)}|$ with ε_j, for all $j = 1, \ldots, k$ concurrently. This would require $O(\log k)$ time on $\lceil k^2/\log k \rceil$ processors. Finally, if $\lceil tk/\log k \rceil$ processors are available for some parameter $1 \le t \le k$, we may perform algorithm 1 in $O((k \log k)/t)$ time by batching $\lceil t \rceil$ consecutive values of j in parallel. In practice, the algorithm needs to examine only a few values of j, so $O(\log k)$ time suffices even with $\lceil k/\log k \rceil$ processors.

Remark 2. If actually $x = 0$, the algorithm can be greatly sped up by testing if $x_j = 0$ in step 2, in which case we may directly pass to $j-1$. Furthermore, stage 3 is not needed unless $x = x_j = 0$ for all j, which can be tested beforehand. Of course, if the only answer needed is "$x = 0$" or "$x \neq 0$", then it suffices to test if all the x_i's are zero and this can be done during the computation of the x_i's.

Remark 3. The costly part of the computation is likely to be the determination of the x_i's. For these reasons, we should try to minimize the number k of moduli m_i involved in the algorithm. This can be done by getting better upper estimates on the magnitude of the output or by using the probabilistic technique of section 4.

Remark 4. The preprocessing table can be made of linear size at the expense of additional operations. Indeed, assume that all the values of k given to the algorithm are less than some value K. Noting that $w_i^{(k)} = w_i^{(K)} \prod_{j=k+1}^{K} m_j \bmod m_i$, we may let $M_k = \prod_{j=k+1}^{K} m_j \bmod m_i$ and store only the tables $w_i^{(K)}$ and M_i for $1 \leq i < K$. Then the $w_i^{(k)}$'s can be computed with k modular multplications in a step 0 of the algorithm above.

3.2 A generalization of Lagrange's method

We will show that Lagrange's method is in fact a particular case of the following method. Let

$$\Sigma^{(0)} = S^{(k)} = \mathrm{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i}\right).$$

This quantity is computed in the first step of algorithm 1. If the computed value of $\Sigma^{(0)}$ is smaller than ε_k, it implies that $\Sigma^{(0)} < 2\varepsilon_k$. Thus, $|x|$ is smaller than $2m\varepsilon_k$. We can then multiply $x_i w_i$ by

$$\alpha_k = \left\lfloor \frac{\frac{1}{2}(1-\varepsilon_k)}{2\varepsilon_k} \right\rfloor,$$

to obtain $(x_i w_i \alpha_k) \bmod m_i$ for all $i = 1, \ldots, k$. This can easily be done by precomputing α_k modulo each m_i. We then compute

$$\Sigma^{(1)} = \mathrm{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i \alpha_k) \bmod m_i}{m_i}\right),$$

and more generally,

$$\Sigma^{(j)} = \mathrm{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i {\alpha_k}^j) \bmod m_i}{m_i}\right),$$

where we assume $\alpha_k \bmod m_i$ precomputed for all $i = 1, \ldots, k$. It is easy to see that the number of iterations in this process is $\lceil \log m / \log \alpha_k \rceil \leq k$, because $|x|$ is no less than 1 and no more than $m^{(k)} \leq 2^{k(b/2+1)}$, and is multiplied by α_k at each iteration. This number is smaller than $\lceil k/2 \rceil + 1$ for all practical purposes. In the implementation, we may assume $x \neq 0$, because this can be tested easily beforehand (see remark above). In this case, we exit necessarily within this number of iterations, hence we do not even need to test for the maximal number of iterations. Therefore, algorithm 2 still performs $\Theta(k^2)$ operations in the worst case, but in practice (on most instances) only k operations of each kind.

This leads to the following algorithm:

Algorithm 2 : Generalized Lagrange's method.
Compute the sign of x knowing $x_i = x \mod m_i$ for all $1 \leq i \leq k$

Precomputed data: m_i, w_i, ε_k, $\alpha_k \mod m_i$, *for all* $i = 1, \ldots, k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for all* $i = 1, \ldots, k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$
Preconditions: $|x| \leq \frac{m^{(k)}}{2}(1 - \varepsilon_k)$ *and* $x \neq 0$

1. *Let* $j \leftarrow -1$, $z_i = x_i w_i^{(j)} \mod m_i$ *for all* $1 \leq i \leq k$
2. *Repeat* $j \leftarrow j + 1$,
   $\Sigma^{(j)} \leftarrow frac\left(\sum_{i=1}^{k} \frac{z_i}{m_i}\right)$
   if $|\Sigma^{(j)}| \leq \varepsilon_k$ *then* $z_i \leftarrow z_i \alpha_k \mod m_i$ *for all* $1 \leq i \leq k$,
   until $|\Sigma^{(j)}| > \varepsilon_k$
3. *If* $\Sigma^{(j)} > 0$ *return* "$x > 0$"
4. *If* $\Sigma^{(j)} < 0$ *return* "$x < 0$"

Remark 5. Algorithm 1 corresponds to a choice of m_j instead of α_k in step j. This simplifies the computation by eliminating one modulus at each iteration, but it performs more iterations. Multiplying by α_k, we perform fewer iterations but each iteration is done with k moduli. This is why we call algorithm 2 a generalization.

Remark 6. To yield the parallel time bounds such as $O(\log k)$ using $\lceil k^2 / \log k \rceil$ processors for algorithm 2, we need to precompute ${\alpha_k}^j \bmod m_i$ for all $i, j = 1, \ldots, k$.

3.3 A probabilistic variant

In the previous algorithm, there can be at most $h_{worst} = \lceil \log(2m^{(k)}\varepsilon_k - 1) / \log \alpha_k \rceil$ iterations. The actual number h_{actual} of iterations is the minimum h that satisfies $|x{\alpha_k}^h| \geq 2m^{(k)}\varepsilon_k$. In the previous algorithm, we find this number by repeatedly incrementing h. In theory we could perform a binary search on h by testing whether $|x{\alpha_k}^h| \geq 2m^{(k)}\varepsilon_k$. Since the value of x is unkown, however, we can only test if $|x{\alpha_k}^h \bmod m^{(k)}| \geq 2m^{(k)}\varepsilon_k$ by using step 2 of the algorithm. If this is detected to hold for some value of h, then necessarily $|x{\alpha_k}^h| \geq 2m^{(k)}\varepsilon_k$ and we may try a smaller value of h. Otherwise, $x\alpha_k^h$ is close to a multiple of $m^{(k)}$, but this is only a probabilistic indication that $|x{\alpha_k}^h| < 2m^{(k)}\varepsilon_k$; we may try nevertheless a greater value of h. We therefore begin with $h = 0$, and then double the value of h until the condition $|x{\alpha_k}^h \bmod m^{(k)}| \geq 2m^{(k)}\varepsilon_k$ is true. Then we perform a binary search for h_{actual} in the range $[0, h]$. Since this range is not guaranteed to contain the value h_{actual}, but does it with high probability, we call this technique *binary search in a probabilistic range.*

Since $2\varepsilon_k$ is much smaller than 1, the probability that, for some fixed h, k, α_k, a random x in the range $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$ satisfies $|x{\alpha_k}^h \bmod m^{(k)}| \geq 2m^{(k)}\varepsilon_k$ but not $|x{\alpha_k}^h| \geq 2m^{(k)}\varepsilon_k$

is extremely small. In fact, if α_k is prime with $m^{(k)}$, it equals the probability that a random y satisfies $|y \bmod m^{(k)}| \geq 2m^{(k)}\varepsilon_k$ but not $|y| \geq 2m^{(k)}\varepsilon_k$. This probability is clearly less than ε_k and is therefore extremely small. The resulting sign, however, is only correct with very high probability. This approach is similar in spirit to that of section 4.2. The speedup is obtained by the fact that only $O(\log k)$ iterations are processed. The resulting algorithm performs only $O(k \log k)$ operations. It may be executed on $\lceil k/\log k \rceil$ processors in parallel time $O(\log^2 k)$.

4 Newton's method

An incremental version of Chinese remainder reconstruction, named after Newton, is described in this section. Its main advantage is that the intermediate computations do not require an *a priori* bound on the magnitude of x, more exactly on the number of moduli needed. This bound is only needed for the deterministic algorithm to stop; in the probabilistic version, no such bound is needed. It also only requires a linear precomputed table.

4.1 The basic method

Let $x^{(j)} = x \bmod m^{(j)}$, for $j = 1, \ldots, k$, so that $x^{(1)} = x_1$ and $x = x^{(k)}$. Let $y_1 = x_1$, and for all $j = 2, \ldots, k$,

$$\begin{aligned} t_j &= w_j^{(j)} = (m^{(j-1)})^{-1} \bmod m_j, \\ y_j &= \left(x_j - x^{(j-1)}\right) t_j \bmod m_j \in \left[-\frac{m_j}{2}, \frac{m_j}{2}\right). \end{aligned}$$

Then (see, e.g., [30, 31]), for all $j = 2, \ldots, k$,

$$x^{(j)} = \left(x^{(j-1)} + y_j m^{(j-1)}\right) \bmod m^{(j)}. \tag{2}$$

Clearly, this leads to an incremental computation of the solution $x = x^{(k)}$ to problem 1; we see below how this can be exploited for an early termination of the interpolation. A further advantage is that all computation can be kept modulo m_j, and no floating-point computation is required, in contrast to sections 3.1 and 3.2 where $S^{(j)}$ or $\Sigma^{(j)}$ are computed. It is obvious, that when $y_j \neq 0$, then the sign of $x^{(j)}$ is the same as the sign of y_j since $|x^{(j-1)}| \leq m^{(j-1)}/2$. If $y_j = 0$, the sign of $x^{(j)}$ is the same as that of $x^{(j-1)}$, for $j \geq 2$, whereas the sign of $x^{(1)} = x_1 = y_1$ is known. If $y_j = 0$ for all j, then this is precisely the case when $x = 0$.

Unrolling equation (2) in the definition of y_j shows that the quantities y_j verify the following Horner-like identity for all $j = 2, \ldots, k$:

$$\begin{aligned} y_j &= \left(x_j - (x^{(j-2)} + y_{j-2} m^{(j-2)})\right) t_j \bmod m_j \\ &\vdots \\ &= \left(x_j - x_1 - m_1(y_2 + m_2(\cdots(y_{j-2} + m_{j-2} y_{j-1})\cdots))\right) t_j \bmod m_j \end{aligned}$$

All the computations are done modulo m_j. Therefore, they can be computed by using modular arithmetic with bit-precision given by the maximum bit-size of the m_j^2. Here it suffices to assume that the absolute value of x is bounded by $m^{(k)}/2$.

Algorithm 3 : Compute the sign of x, knowing $x \mod m_i$, by Newton's incremental method

Precomputed data: *m_j, t_j, for all $1 \le j \le k$*
Input: *integers k and $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ for all $i = 1, \ldots, k$*
Output: *sign of x, where x is the unique solution of $x_i = x \mod m_i$ in $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$*
Precondition: *none.*

1. *Let $y_1 \leftarrow x_1$, $j \leftarrow 1$. Depending on whether y_1 is negative, zero or positive, set s to $-1, 0$ or 1, respectively.*
2. *Repeat $j \leftarrow j + 1$,*

$$y_j \leftarrow (x_j - x_1 - m_1(y_2 + m_2(\cdots(y_{j-2} + m_{j-2}y_{j-1})\cdots)))\, t_j \mod m_j,$$

until $j = k$. For every j, set s to 1 or -1, if y_j is positive or negative, respectively.
3. *Depending on whether s is -1, 0, or 1, return "$x < 0$", "$x = 0$", or "$x > 0$", respectively.*

Remark 7. As in remark 1, we can test beforehand if all $x_i = 0$, which is precisely the case when $x = 0$.

Lemma 4.1 *Algorithm 3 computes the sign of x knowing its residues x_i using at most $\frac{k(k-1)}{2}$ f.p. modular multiplications, $\frac{k(k-1)}{2}$ modular additions, and k f.p. comparisons.*

Proof. For every $j = 2, \ldots, k$, there are $j-1$ modular additions and multiplications. There is one comparison for each $j = 1, \ldots, k$. □

Algorithm 3 requires k iterative steps, so its parallel time cannot be decreased below $\Omega(k \log k)$. Nevertheless the algorithm can be implemented in $O(k \log k)$ time on $\lceil k/\log k \rceil$ processors, assuming each b-bit f.p. operation takes constant time.

To compare with algorithm 1, realistically assume that a modular addition is equivalent to 3/2 f.p. additions and one comparison, on the average. Then, algorithm 1 requires $\frac{k(k-1)}{2}$ f.p. divisions (which are essentially multiplications with precomputed reciprocals) more than algorithm 3, whereas the latter requires $\frac{k(k-1)}{4}$ extra f.p. additions and $\frac{k(k-1)}{2}$ additional comparisons.

The principal feature of this approach, based on Newton's formula for recovering x, is its incremental nature. This may lead to faster termination, before examining all k moduli. Informally, this should happen whenever the magnitude of x is significantly smaller than $m^{(k)}/2$, in which case we would save the computation required to obtain x_j for all larger j. This saves a significant amount of computation if termination occurs earlier than the static

bound indicated by k. A quantification of this property in the case of convex hulls can be found in [17].

Another merit of this approach is that the size of the precomputed table is only linear in k (as opposed to quadratic for algorithms 1 and 2). For very large values of k, this can be the method of choice.

4.2 A probabilistic variant

We propose below a probabilistic variant of algorithm 3 which, moreover, removes the need of an *a priori* knowledge of k. Step 2 is modified to include a test of y_j against zero. Clearly, $y_j = 0$ precisely when $x^{(j)} = x^{(j-1)}$. Then we may deduce that $x^{(j)} = x^{(k)} = x$, with a very high probability, and terminate the iteration.

Algorithm 4 : Yield earlier termination of algorithm 3 for absolutely smaller input. Algorithm 3 is modified exactly as shown.

Input: *integers* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for* $i = 1, \ldots$ *as required in the course of the algorithm; no need for* k
Output: *sign of* x *with very high probability*

2. Terminate the loop also if $y_j = 0$

By lemma 3.1 of [17], this algorithm fails with probability bounded by $(k-2)/m_{\min}$, where

$$m_{\min} = \min\{m_1, m_2, \ldots, m_k\}.$$

For $k \leq 12$, $m_{\min} \geq 2^{25}$, the error probability is less than 10^{-6}. A more careful analysis can reduce this probability to $O((k-2)/m_{\min}^{\ell})$ by exploiting the correlation of failure at different consecutive stages. For experimental support of this claim, we refer to [17]. The only modification to the algorithm is to replace step 2 by step 2′ below:

2'. Terminate the loop also if $y_j = 0, \ldots, y_{j-\ell} = 0$

5 Filtered residue number systems (F-RNS)

Given a number x such that $|x| \leq m^{(k)}/2$, we call $x \simeq (x_1, \ldots, x_k)$ a *k-modular representation* of x. Using the techniques of the previous sections, we can reduce the number k in the representation of x to its minimum: we call this process *normalization*. In the next section, we explain the technique of modular inference, which extends a k-modular representation into a k'-modular representation, $k' > k$. For filtering as well as for getting a sharp upper bound on the number of moduli needed to represent a number, we can maintain for each number an approximate f.p. representation. This is done in section 5.2. Finally, in section 5.3, we show how to perform all arithmetic operations in hybrid symbolic-numeric RNS which entails a small number of single precision f.p. operations. The small overhead and the simplicity of our schemes make it very well suited for implementation.

5.1 Modular inference

Given x_j, with $|x_j| \leq m_j/2$ for all $j = 1, \ldots, k$, we know that there exists a unique x such that $|x| \leq m^{(k)}/2$ and that $x_j = x \bmod m_j$ for all $j = 1, \ldots, k$. In some computations where x is involved, we may need furthermore $x_i = x \bmod m_i$ for $i = k+1, \ldots, k'$.

Problem 2 *Let k, k', b, $m_1, \ldots, m_{k'}$ denote positive integers, $m_1, \ldots, m_k$ being pairwise relatively prime, such that $k < k'$ and $m_i \leq 2^{b/2+1}$. Let x be an integer whose magnitude is smaller than $\lfloor (m/2)(1 - 2^{-b}) \rfloor$. Given $x_i = x \bmod m_i$ for $i = 1, \ldots, k$, compute $x_i = x \bmod m_i$ for $i = k+1, \ldots, k'$.*

We will solve this following problem, which we call *modular inference*, by avoiding the full reconstruction of x and using only single precision and modular operations.

Newton's method. Our first technique is based on equation (2), which implies that

$$x = \left(\cdots \left(y_k m_{(k-1)} + y_{k-1}\right) m_{(k-2)} + \cdots + y_2\right) m_1 + y_1.$$

The values of the y_j's are computed as a byproduct of algorithm 3. Using this expression, we may compute $x \bmod m_i$ very easily, for any $i = k+1, \ldots, k'$, by computing each term in the sum above modulo m_i and then computing the sum x modulo m_i. We therefore obtain the following algorithm.

Algorithm 5 : Compute $x \bmod m_i$ knowing $x \bmod m_j$, for all $1 \leq j \leq k < i \leq k'$

Precomputed data: *m_j, $m_i \bmod m_j$, for all $1 \leq j \leq k < i \leq k'$*
Input: *integers k, k' and $x_i \in \left[-\frac{m_j}{2}, \frac{m_j}{2}\right)$ for all $j = 1, \ldots, k$*
Output: *$x \bmod m_i$ for all $i = k+1, \ldots, k'$, where x is the unique solution of $x_j = x \bmod m_j$ $(1 \leq j \leq k)$ in $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$*
Precondition: *None.*

1. *Compute y_j for all $j = 1, \ldots, k$ as in algorithm 3.*
2. *For all $i = k+1, \ldots, k'$, do*

$$x_i \leftarrow \left(\left(\cdots \left(y_k m_{(k-1)} + y_{k-1}\right) m_{(k-2)} + \cdots + y_2\right) m_1 + y_1\right) \bmod m_i.$$

The following lemma is immediate.

Lemma 5.1 *Algorithm 5 computes a k'-modular representation of x from a k-modular representation with $k(k'-k)$ modular multiplications and $(k-1)(k'-k)$ modular additions, plus all the operations given in lemma 4.1.*

As a comparison, using multiple precision arithmetic would involve $\Theta((k'+k)k)$ single precision operations: first $\Theta(k^2)$ to recover the exact value of x and then $\Theta(k)$ operations for each $i = k+1, \ldots, k'$ to perform the Euclidean division by m_i. In theory, it can be accelerated by using the asymptotically fastest multiplication algorithm in $O((k'+k)\log^2(k'+k)\log\log(k'+$

k)) time [1], albeit with an enormous overhead. More practically, the $O((k'+k)k)$ time multiprecision approach is asymptotically equivalent to our above algorithm.

If the y_j's are not known, however, step 1 of this algorithm requires to perform a quadratic number of operations even if only a single extra modulo is needed ($k' = k+1$). This is improved by the following algorithm.

Lagrange's method. Equation (1) implies that, for some integer X_k,

$$x = \sum_{j=1}^{k} ((x_j w_j) \bmod m_j)\, v_j - X_k m^{(k)}.$$

The key observation is that X_k may be computed by step 2 of algorithm 1, because X_k is the integral part of the first sum whose fractional part is $S^{(k)}$. Moreover, X_k is very small (less than k) and for all practical purposes is much smaller than all the m_i's, $i \leq k'$. Hence, $X_k \bmod m_i = X_k$. Assume we have precomputed $v_{i,j} = v_j \bmod m_i$ for $1 \leq j \leq k < i \leq k'$.

Algorithm 6 : Compute $x \bmod m_i$ knowing $x \bmod m_j$, for all $1 \leq j \leq k < i \leq k'$

Precomputed data: *m_j, $v_{i,j}$, $m^{(k)} \bmod m_i$, for all $1 \leq j \leq k < i \leq k'$*
Input: *integers k, k' and $x_i \in \left[-\frac{m_j}{2}, \frac{m_j}{2}\right)$ for all $j = 1, \ldots, k$*
Output: *$x \bmod m_i$ for all $i = k+1, \ldots, k'$, where x is the unique solution of $x_j = x \bmod m_j$ $(1 \leq j \leq k)$ in $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$*
Precondition: $|x| \leq \frac{m^{(k)}}{2}(1-\varepsilon_k)$

1. *Compute $z_j \leftarrow (x_j w_j) \bmod m_j$ for all $j = 1, \ldots, k$.*
2. *Compute X_k by performing step 2 of algorithm 1 for $j = k$.*
3. *For all $i = k+1, \ldots, k'$, do*

$$x_i = \left(\sum_{j=1}^{k} z_j v_{i,j} - X_k m^{(k)} \right) \bmod m_i.$$

We now count the number of arithmetic operations.

Lemma 5.2 *Algorithm 6 computes a k'-modular representation of x from a k-modular representation with at most $k(k'-k+1)-1$ modular additions, $(k'-k+2)(k+1)-2$ modular multiplications, and k f.p. divisions.*

Proof. Step 1 involves k modular multiplications, while step 2 performs k modular multiplications and divisions and $k-1$ f.p. additions. As for step 3, it performs $k'-k$ iterations, each of which computes $k+1$ modular multiplications and k modular additions. □

This method based on Lagrange's formula is asymptotically faster than all known methods when $k'-k = o(k)$. For instance, recovering a single additional modulo x_{k+1} requires

time $O(k \log^2 k \log\log k)$ and a huge overhead with the fastest known multiplication algorithms [1], while more practical methods would run in time $O(k^2)$. In this case, our method requires only $3k-1$ modular operations, k f.p. divisions, and very little overhead.

5.2 Computing the closest f.p. representation

We use the notation of section 2. Algorithms 1 and 3 compute in fact a good f.p. approximation of the number x represented by its moduli x_i. Unfortunately, this is not enough as we sometimes need to compute the *closest* f.p. approximation $\widetilde{x}$ of x, that is, with absolute error less than $\frac{1}{2}\mathrm{ulp}(x)$ (see section 2). This is called exact rounding. and extends to arbitrary precision the exact rounding provided for elementary operations by the IEEE 754 double precision standard. Exact rounding has many merits. It is also useful in computational geometry for *renormalization* (see [33] and references therein), such as rounding to a grid the vertices of a planar map obtained by various means (line intersections, circumcenters, etc.). For filtered RNS, when $\widetilde{x}$ and $\widetilde{y}$ differ, comparing x and y can safely be decided by comparing $\widetilde{x}$ and $\widetilde{y}$ without calling for algorithms 1 or 3. We therefore want to solve the following problem.

Problem 3 *Let k, b, $m_1, \ldots, m_k$ denote positive integers, $m_1, \ldots, m_k$ being pairwise relatively prime, such that $m_i \leq 2^{b/2+1}$. Let x be an integer whose magnitude is smaller than $\lfloor (m/2)(1-\varepsilon_k) \rfloor$. Compute a f.p. number $\widetilde{x}$ such that $|\widetilde{x} - x| \leq \frac{1}{2}ulp(x)$ by using only f.p. operations with b-bit precision.*

As with modular inference, Newton's method is applicable but requires an inherently quadratic number of operations (we must compute all the y_j's). But Lagrange's method seems to yield exact rounding with $O(k)$ operations in practice, unless $x - \widetilde{x}$ is extremely close to $\frac{1}{2}\mathrm{ulp}(x)$. Indeed, the computed value $R^{(k)}$ of $S^{(k)}m^{(k)}$ approximates x with an absolute error $E^{(k)} = x - R^{(k)}$ bounded by $(\varepsilon_k + 2^{-b-1})m^{(k)}$ (see section 3.1). The key idea is that we can compute an exact $(k-1)$-representation of this error since $E^{(k)} \bmod m_j = (x_j - R^{(k)}) \bmod m_j$. We can again compute an approximation $R^{(k-1)}$ of $E^{(k)}$, by applying Lagrange's method on $k-1$ moduli and computing $S^{(k-1)}m^{(k-1)}$ with an absolute error $E^{(k-1)}$ bounded by $(\varepsilon_{k-1} + 2^{-b-1})m^{(k-1)}$. Recursively, for a decreasing $j = k-1, \ldots, 1$, we can compute a $(j-1)$-representation of $E^{(j)}$ by $E^{(j)} \bmod m_i = (R^{(j+1)} - R^{(j)}) \bmod m_i$ for each $i = 1, \ldots, j$. An approximation $R^{(j-1)}$ of $E^{(j)}$ is obtained by Lagrange's method on $j-1$ moduli with absolute error $E^{(j-1)}$ bounded in magnitude by $(\varepsilon_{j-1} + 2^{-b-1})m^{(j-1)}$. For any $j = k-1, \ldots, 0$, we have

$$x = R^{(k)} + \cdots + R^{(j)} + E^{(j)}.$$

Since the non-zero $R^{(i)}$'s are decreasing by a factor $\varepsilon_k < 1/2$, a good way to compute their sum is to perform the additions in the order $R^{(k)} + (R^{(k-1)} + (\cdots + (R^{(j+1)} + R^{(j)}) \cdots))$. With b-bit f.p. precision, the result Z_j approximates $\sum_{i=j}^{k} R^{(i)}$ with an absolute error bounded by $\mathrm{ulp}(Z_j)$. Moreover, truncating the leading bits of $R^{(i)}$ to $\mathrm{ulp}(Z_j)$ yields a sum z_j truncated to $\mathrm{ulp}(Z_j)$ such that $Z_j + z_j$ approximates $\sum_{i=j}^{k} R^{(i)}$ with an absolute error bounded by

$\mathrm{ulp}(z_j)$. (Note that truncating $R^{(i)}$ to $\mathrm{ulp}(Z_j)$ can be achieved by computing $(R^{(i)}+Z_j)-Z_j$ with b-bit f.p. precision.) Then $Z_j + z_j$ approximates x with an absolute error bounded by

$$\mathrm{ulp}(z_j) + |E^{(j)}| \leq \mathrm{ulp}(z_j) + (\varepsilon_j + 2^{-b-1})m^{(j)}.$$

If $Z_j + z_j$ truncated to $\frac{1}{2}\mathrm{ulp}(Z_j + z_j)$ is distant from $\frac{1}{4}\mathrm{ulp}(Z_j + z_j)$ by at least $\mathrm{ulp}(z_j) + (\varepsilon_j + 2^{-b-1})m^{(j)}$, then $Z_j + z_j$ can be rounded to the closest f.p. representable number and yields the desired $\widetilde{x}$.

A minor difficulty arises because it is hard to compute $R^{(j)} \bmod m_i$: in the worst case, this may take $\left\lfloor \frac{1}{b} \log R^{(k)} \right\rfloor$ f.p. divisions. However, this is one single instruction in the IEEE 754 standard and we will account for it as a single instruction. (Note that although $R^{(k)}$ is a multiple of $m(k)$ by $S^{(k)}$, $S^{(k)}$ is not an integer, and there is roundoff error in computing the product $S^{(k)}m^{(k)}$, so we cannot take advantage of this to compute $R^{(j)} \bmod m_i$.)

This leads to the following algorithm.

Algorithm 7 : Compute $\widetilde{x}$ knowing $x_i = x \bmod m_i$

Precomputed data: m_j, $\widetilde{m^{(j)}}$, $w_i^{(j)}$, η_j, *for all* $1 \leq i \leq j \leq k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, *for all* $1 \leq i \leq k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$
Precondition: $|x| + \eta_k \leq \frac{m^{(k)}}{2}$

1. *Let* $j \leftarrow k$, $R^{(k)} \leftarrow \mathit{frac}\left(\sum_{i=1}^{k} \frac{x_i w_i^{(k)} \bmod m_i}{m_i}\right) m^{(k)}$,
2. *Repeat* $j \leftarrow j - 1$,
   $x_i \leftarrow (x_i - R^{(j+1)}) \bmod m_i$ *for all* $i = 1, \ldots, j$
   $R^{(j)} \leftarrow \mathit{frac}\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i}\right) m^{(j)}$
   $Z_j \leftarrow R^{(k)} + (R^{(k-1)} + (\cdots + (R^{(j+2)} + R^{(j+1)}) \cdots))$
   $z_j \leftarrow R^{(k)} + (R^{(k-1)} + (\cdots + (R^{(j+2)} + R^{(j+1)}) \cdots))$ *truncated to* $\mathit{ulp}(Z_j)$
   $x' \leftarrow (Z_j + z_j)$ *truncated to* $\frac{1}{2}\mathit{ulp}(Z_j + z_j)$
   until $j = 0$ *or* $\left|x' - \frac{1}{4}\mathit{ulp}(Z_j + z_j)\right| > \mathit{ulp}(z_j) + (\varepsilon_j + 2^{-b-1})m^{(j)}$
3. *return* $Z_j + z_j$

As for algorithm 1, algorithm 7 requires $\Theta(k^2)$ operations in the worst case. In practice, however, algorithm 7 terminates at $j = k - 2$, and uses only $O(k)$ operations.

Remark 8. The ideas of section 3.2 also apply to this case, but multiplying by α_k introduces additional errors, so instead we multiply by a value $\beta_k = 2^{\lfloor \log \alpha_k \rfloor}$.

Remark 9. We can also obtain the closest f.p. approximation of a quotient $z = x/y$ where x and y are given by their k-modular representations. Indeed, when computing z' by performing $\widetilde{x}/\widetilde{y}$ with b-bit precision, only the last (at most) three bits are incorrect. The correct bits can be determined by at most three binary searches: if z' is expressed exactly as u/v (for instance, we could take $v = 1/\mathrm{ulp}(z)$ if z' is not an integer), then we can compute modular representations of u and v and compare z' with z by computing the sign of $vx - uy$ with the help of algorithm 1 or algorithm 3.

5.3 Filtered RNS

Assume we are given a k-modular representation of a number x and a k'-modular representation of y. Without loss of generality, we assume that $k' \leq k$.

To obtain the closest b-bit f.p. representation of x, we apply algorithm 7, and so the closest f.p. approximation is obtainable with $O(k^2)$ single-precision f.p. operations. In theory, we could maintain the closest f.p. representation of a number under all the arithmetic operations in RNS. In practice, it suffices that the approximation be with relative error $1/2$ to get the sign, and so we need only "refresh" the approximation when the result of an operation is much smaller than the summands.

The cost of normalization is $O(k^2)$ if we use algorithm 3, but is only $O(ks)$ if we use algorithm 1, where s is the *slack* of the representation, the number of iterations in step 2 of algorithm 1; if we maintain the closest f.p. approximation of a number, it takes usually no operation and at most $s = 1$ iteration of algorithm 1 to get the normalized representation.

To represent the sum $x + y$, we first compute a k'-modular representation of x using algorithm 6. A representation of the sum $x + y$ can readily be obtained by adding in each finite field. Overflow can be efficiently determined by using the f.p. approximations, and in this case one inference with k' modular multiplications and additions provide the additional modulo needed to represent the result. Overall, the addition is done by using at most ??? modular additions, ??? modular multiplications, one f.p. addition on the approximations, and one f.p. comparison. In particular, if $k' = k$, the addition requires at most $4k - 1$ modular operations and k f.p. divisions.

A representation of the product xy is obtained readily by constructing $(k + k')$-modular representation of both x and y, multiplying in each finite field, and normalizing the result if a normal representation is needed. (Note that this last step may be needed even if the representations of x and y are normal; the correct number of moduli needed to represent xy may be obtained directly from f.p. approximation of xy.) This involves at most $2kk' + k + k' - 2$ f.p. modular additions, $2(kk' + k + k' - 1)$ modular multiplications, and k f.p. divisions.

Division of x by a single precision number m such that $|m| < 2^{b/2+1}$ can be done using the inference of section 5.1 in $O(k)$ time (for both quotient and remainder of the division by m). Division of y by x can be performed by using methods similar to ours as described by Hung and Parhami [28].

Lastly, comparison can be handled by comparing the f.p. approximations, and if necessary by using the recursive relaxation of the moduli. In this case, let us first assume that

x and y are given in normal form. It is easy to solve the comparison if $k \neq k'$. Otherwise, $O(k^2)$ single precision operations may be needed to determine the result of the comparison. Note that in practice, we expect that the comparison will resolve with $O(k)$ operations. Note also that the numbers k and k' of moduli in the normal representation is given directly by the f.p. approximation, so there is no need to actually perform the normalization.

We summarize this discussion in the following theorem.

Theorem 5.3 *Let x (resp. y) be given by a normalized k-modular (resp. k'-modular) representation, $k' > k$. In filtered RNS, additions can be performed by using $O(k(k'-k)+k+k')$ modular operations, multiplications by using $O(kk'+k+k')$ operations, divisions by a small integer by using $O(k)$ operations, and comparisons by $O(1)$ operations is $k \neq k'$, and $O(k^2)$ operations if $k = k'$.*

In an arithmetic big number package, additions and comparisons require $O(k+k')$ operations, while multiplications require $O((k+k')\log(k+k')\log\log(k+k'))$ operations with huge overhead, and are more commonly implemented with $O((k+k')^2)$ operations. Filtered RNS perform all these operations with little overhead in time $O(k+k')$, except when modular inference is needed. But modular inference can be amortized over all the computations since it needs only be performed once for each number. So RNS used with our techniques provide no slower, and sometimes faster multiplications, at the cost of slightly slower (and in practice, comparable) additions and comparisons. They are practical and can easily be implemented with low overhead. Moreover, they are easily parallelizable.

6 Applications

Our solutions to problem 1 have many applications. Below we focus on three major areas, namely computation with real algebraic numbers, exact geometric algorithms, and the ubiquitous question of determinant sign. Additional applications include numeric algorithms for reducing the solution of general systems of analytic equations to sign evaluation [40], deciding the theory of the reals [10, 4], geometric theorem proving [37], and manipulating sums of radicals [2].

6.1 Real algebraic numbers

Being able to compute efficiently with algebraic numbers is important but also necessary in a variety of computer algebra applications, as well as when calculating over the reals. In particular, it is a fundamental operation when computing with algebraic numbers, which is a robust way to treat real numbers, and in general when numeric computation does not offer the required guarantees.

The critical operation is deciding the sign of a multivariate polynomial expression with rational coefficients on a set of points. We will show how our solution can be applied to the manipulation of real algebraic numbers. We refer to [13, 34] for a comprehensive review of the algebraic concepts involved.

A popular paradigm for manipulating algebraic numbers is the use of Sturm sequences. Given two polynomials P and Q in $\mathbb{Z}[X]$, $\deg(P) \geq \deg(Q)$, we consider a sequence $\Sigma = (P_0, P_1, \ldots, P_m)$ of polynomials such that $P_0 = P$, $P_1 = Q$, $\deg(P_i) < \deg(P_{i-1})$ for all $i = 1, \ldots, m$. We will assume that P and Q are square-free and do not vanish at a or b. Let $\mathrm{Var}_{P,Q}(a)$ be the number of sign changes of the sequence $\Sigma(a) = (P_0(a), P_1(a), \ldots, P_m(a))$, and define $\mathrm{Var}_{P,Q}[a,b] = \mathrm{Var}_{P,Q}(a) - \mathrm{Var}_{P,Q}(b)$. Sturm sequences have the property that

$$\mathrm{Var}_{P,Q}[a,b] = \sum_{\gamma} \mathrm{sign}(P'(\gamma)Q(\gamma)),$$

where γ ranges over all roots of P in $[a,b]$. Of special interest is the case where Q is the derivative P' of P. In this case, we write $\mathrm{Var}_P[a,b]$ for $\mathrm{Var}_{P,P'}[a,b]$, and this number equals the number of roots of P in $[a,b]$.

It turns out that the coefficients of the P_i's grow very fast, even for simple P and Q. This phenomenon is well known in computer algebra, and seems to require the computations over very large integers. One popular alternative is modular arithmetic. The bottleneck of this approach (at least in theory) is the computation of $\mathrm{Var}_{P,Q}[a,b]$, which involves many sign reconstructions. The recursive relaxation of the moduli is ideally suited because the exact value of $P_i(a)$ is never needed, but only its sign. Therefore, once the sequence Σ is computed in the several finite fields, we may evaluate $\Sigma(a)$ in each finite field and apply algorithm 1 to compute the corresponding sign sequence and finally $\mathrm{Var}_P(a)$.

We examine the complexity of our algorithm for computing the sign sequence corresponding to $\Sigma(a)$ at some rational number a. Let n denote the maximum degree of P and Q, L denote the maximum size of the coefficients of the input polynomials P, Q, and l the sum of the sizes of the numerator and denominator of a. The degrees are decreasing so the length of the sequence is $m \leq n$. As shown in [13], the time to compute the sequence Σ is $O(n^4(L + \log n)^2)$, and the coefficient of the P_i's are bounded by $2^{2n(L+\log n)}$. Hence $P_i(a)$ is bounded by

$$|P_i(a)| = n2^{2n(L+\log n)}2^{ln},$$

and therefore $O(n(L + l + \log n))$ moduli are sufficient. By using algorithm 1, we correctly retrieve the sign of $P_i(a)$ in time $O(n^2(L+l+\log n)^2)$, for each $i = 0, \ldots, m$. If the sequence is known in each finite field, the computation of the sign sequence corresponding to $\Sigma(a)$ can therefore be done in time $O(n^3(L + l + \log n)^2)$ in the worst case. We summarize this in the following theorem:

Theorem 6.1 *Knowing the Sturm sequence Σ modulo each m_i, $i = 1, \ldots, k$, where $k = O(n(L + l + \log n))$, one can compute $Var_P(a)$ in time $O(n^3(L + l + \log n)^2)$.*

The performance given in the above theorem is in the worst case, however, and in practice, algorithm 1 will run in time $O(k) = O(n(L+l+\log n))$. This lowers the expected complexity of the computation of $\mathrm{Var}_P(a)$ to $O(n^2(L + l + \log n))$ in practice.

As an application of those ideas, we show how to manipulate algebraic numbers. An algebraic number α can be represented symbolically by a square-free polynomial $P \in \mathbb{Z}[X]$

and an interval $I = [a, b]$, such that α is the only root of P in $[a, b]$ (with multiplicity at least but not necessarily 1). Such an interval can be found by evaluating Var_P at $O(n(L+\log n))$ points [13]. Moreover, in this context, separation bounds imply that $l = O(n(L + \log n))$. The total time of the root isolation procedure is therefore $O(n^6(L + \log n)^3)$. The expected cost is therefore dominated by the sign computations. Practically, however, this cost is expected to be $O(n^4(L + \log n)^2)$, which is the same as the cost of the computation of the Sturm sequence.

To compare two algebraic numbers $\alpha \cong (P, I)$ and $\beta \cong (Q, J)$, we may first assume that they both lie in $I \cap J = [a, b]$, otherwise the comparison can be performed on the intervals. (This assumption can be checked by evaluating Var_P at the endpoints of J and Var_Q at the endpoints of I.) Then (see [13]), $\alpha \geq \beta$ if and only if

$$\mathrm{Var}_{P,Q}[a, b] \cdot (P(a) - P(b)) \cdot (Q(a) - Q(b)) \geq 0.$$

The expensive part of this computation is therefore the computation of $\mathrm{Var}_{P,Q}[a, b]$, which can be done in time $O(n^4(L + \log n)^2)$ for the computation of the Sturm sequence and $O(n^3(L+l+\log n)^2)$ for the sign determinations. Practically, the cost of the sign computation is negligible compared to the cost of the computation of the Sturm sequence.

Extension to intersections of algebraic curves can be done in much the same fashion, using multivariate Sturm theory; see [34] and the references therein. It has been applied in the context of solid modeling by [29] who use modular arithmetic with a *bignum* library for the sign reconstruction.

6.2 Exact geometric predicates

Exact geometric predicates are the most general way to provide robust implementations of geometric algorithms [16, 20, 44, 18]. For instance, orientation and in-circle tests or the comparison of segment intersections, can all be formulated as deciding the sign of a determinant. Before studying the latter question in its own right, we survey several problems in computational geometry which can make use of our algorithms to achieve robustness and efficiency.

Modular arithmetic becomes increasingly interesting when the geometric tests (e.g. determinants) are of higher dimension and complexity. They are central in, notably, convex hull computations: this is a fundamental problem of computational geometry and of optimization for larger dimensions. Computing Voronoi diagrams of points reduces to convex hulls in any dimension, but is mostly done in dimensions 2 and 3. Nevertheless, the sweepline algorithm in 2 dimensions involves tests of degree 20 and modular arithmetic can be of substantial help, in conjunction with arithmetic filters [21]. For Voronoi diagrams of segments, the tests become of even higher degree and complexity [9], and f.p. computation is likely to introduce errors, so exact arithmetic is often a must.

Even for small dimensions, the nature of the data may force the f.p. computation to introduce inconsistencies, for instance, in planarity testing in geometric tolerancing [43]. Here, one must determine if a set of points sampling a plane surface can be enclosed in a

slab whose width is part of the planarity requirements. The computation usually goes by computing the width of the convex hull, and the data is usually very flat, hence prone to numerical inaccuracies.

In geometric and solid modeling, traditional approaches have employed finite precision floating point arithmetic, based on bounds on the roundoff errors. Although certain basic questions in this domain are now considered closed, there remain some fundamental open problems, including boundary computation [26]. Tolerance techniques and symbolic reasoning have been used, but have been mostly restricted to polyhedral objects; their extension to curved or arbitrary degree sculptured solids would be complicated and expensive. More recently, exact arithmetic has been proposed as a valid alternative for generating boundary representations of sculptured solids, since it guarantees robustness and precision even for degenerate inputs at a reasonable or negligible performance penalty [29]. One key component is the correct manipulation of algebraic numbers (see the previous section).

6.3 Sign of the determinant of a matrix

As mentioned, computing the sign of a matrix determinant is a basic operation in computational geometry, applied to many geometric tests (such as orientation tests, in-circle tests, comparing segment intersections) [12, 3]. Sometimes, the entries to the determinant are themselves algebraic expressions. For instance, the in-circle test can be reduced to computing a 2×2 determinant, whose entries have degree 2 and thus require $2b + O(1)$-bit precision to be computed exactly [3]. Computing these entries by using modular arithmetic enables in-circle tests with b-bit precision while still computing exactly the sign of a 2×2 determinant.

To compute an $n \times n$ determinant modulo m_k, we may use Gaussian elimination with a single final division. At step $i < n$ of the algorithm, the matrix is

$$\begin{pmatrix} \vdots & \dots & \dots \\ 0 & \alpha_{i,i} & \dots \\ \vdots & \vdots & \vdots \\ 0 & \alpha_{n,i} & \dots \end{pmatrix}$$

and we assume that the pivot $\alpha_{i,i}$ is invertible modulo m_k. Then we change line L_j to $\alpha_{i,i}L_j - \alpha_{j,i}L_i$ for all $j = i+1, \dots, n$. At step n of the algorithm, we multiply the coefficient $\alpha_{n,n}$ by the modular inverse of the product $\prod_{i=1}^{n-1} \alpha_{i,i}^{n-i}$. This gives us the value of the determinant modulo m_k. Note that the same method but with non-modular integers and a final division would have involved exponentially large integers and several slow divisions at each step. Nevertheless, it is only the range of the final result that matters for modular computations. This shows a big advantage of modular arithmetic over other multiprecision approaches.

The pivots should be invertible modulo m_k. If m_k is prime, the pivot simply has to be non-zero modulo m_k. The algorithm may be also implemented if m_k is a power of a

prime, or if m_k is the product of two primes. This would be desirable mainly for taking $m_k = 2^{b_k}$ for which modular arithmetic is done naturally by integer processors, though here, special care must be taken about even output. Other choices of m_k do not seem to bring any improvement.

With IEEE double precision ($b = 53$), we choose moduli smaller than 2^{27}, so that $2(\frac{m_k}{2})^2 \leq 2^{53}$: Gaussian elimination intensively uses $(ps - qr)$-style operations; here we may apply one final modular reduction, instead of two for each product before subtracting.

This algorithm performs $O(n^3)$ operations for each modulus m_i. Assume that the entries are integers smaller than 2^b. With Hadamard's determinant bound and m_k greater than $2^{b/2}$, only $k = \left\lceil n(2 + \frac{\log n}{b}) \right\rceil$ finite fields need to be considered. Hence the complexity of finding the sign of the determinant is $O(n^4 \log n)$ single precision operations, when the entries are b-bits integers.

More generally, when the entries are integers of bit-length L, we have to take into account the computation of these n^2 entries modulo m_i, for $i = 1, \ldots, k$. In this case, Hadamard's bound yields $k = \left\lceil n\frac{2L + \log n}{b} \right\rceil$. Each computation amounts to computing the remainder of the division of an L-bit integer by a single-precision integer, in time $O(L)$, for a global cost of $O(n^3 L(\log n + L))$, which can be sped up to approximately $O(n^3(\log n + L)(\log n + \log L))$ as shown in [1], by using divide-and-conquer. Hence, the entire computation takes time $O(n^2(n^2 + L)(\log n + L))$, where b is considered as a constant.

To summarize:

Theorem 6.2 *The algorithm described above computes the sign of a $n \times n$ determinant whose entries are integers of bit-length by using $O(n^2(n^2 + L)(\log n + L))$ single precision operations.*

Using the algorithm of Bareiss for this problem yields a bound $O(n^3 M(n(\log n + L))$, where $M(p)$ is the number of operations to compute the product of two p-bit integers. In practice, we almost always have $L = O(n)$. Using multiplication in time $M(p) = p \log p \log\log p$ yields a slightly worse bound than given in the theorem, albeit with a huge overhead. More practically, using multiplication in time $M(p) = O(p^2)$ yields an order of magnitude slower. Our algorithm is easy to implement and entails little overhead. This is also corroborated by the practical study of section 7.

On a $O(n^3 \log n)$-processor machine, the time complexity drops to $O(n)$, if we use customary parallelization of the Gaussian elimination routine for matrix triangulation (cf. [24]), which gives us the value of the determinant. (We apply this routine in modular arithmetic, with simplified pivoting, concurrently for all m_i's.) Theoretically, substantial additional parallel acceleration can be achieved by using randomization [5, ch. 4], [36], yielding the time bound $O(\log^2 n)$ on $\lceil n^3 \log n \rceil$ arithmetic processors, and the processor bound can be decreased further to $O(n^{2.376})$, by applying asymptotically fast algorithms for matrix multiplication.

7 Experimental results

We present several benchmark results of our implementations in C of the described methods for computing the sign of a determinant, and compare them with different existing packages.

- Method FP is a straightforward f.p. implementation of Gaussian elimination.
- Method MOD is an implementation of modular Gaussian elimination as described in section 6 using our recursive relaxation of the moduli (version 2.0).
- Method PROB is the probabilistic variant described in section 4.2, that fails to give the correct result with a probability bounded by 2^{-50}.
- Method GMP is an implementation using the GNU Multiprecision Package (version 2.0.2), of Gaussian elimination for dimension lower than 5, and of Bareiss' extension of Gaussian elimination for higher dimensions.
- Method LEDA uses the routine `sign_of_determinant(integer_matrix)` of Leda [9] (version 3.5).
- Method LN [21] provides a very fast implementation in dimensions up to 5 but was not available to us in higher dimensions.

Of the other methods available, the lattice method of [6] has not yet been implemented in dimensions higher than 6, and both the lattice method and our implementation of Clarkson's method [12] are limited in the bit size of the entries.

Among the methods that guarantee exact computation, our implementations are at least as efficient as the others, and for certain classes of input they outperform all available programs. Furthermore our approach applies to arbitrary dimensions, whereas methods that compute a f.p. approximation of the determinant value are doomed to fail in dimensions higher than 15 because of overflow in the f.p. exponent.

All tests were carried out on a 200MHz Sun UltraSparc-1 workstation. Each program is compiled with the compiler that gives best results. Each entry in the following tables represents the average time of one run in microseconds, with a maximum deviation of about 10%. We concentrated on determinant sign evaluation and considered three classes of matrices: random matrices, whose determinant is typically away from zero, in table 1, almost-singular matrices with single-precision determinant in table 2, and lastly singular matrices with zero determinant in table 3. The coefficients are integers of bit-size 53.

Our results suggest that our approach is comparable, and for certain classes of input significantly faster than the examined alternatives that guarantee exact results. The running times are displayed in tables 1–3. (For small dimensions, specialized implementations can provide an additional speedup for nearly all methods.) Although full reconstruction of the determinant value may take some time, our method to determine the sign is negligible as can been seen from the difference with random (where it is linear) and null (where it is quadratic) determinants. Our code is reasonably compact and easy to maintain. As an

n	FP	MOD	PROB	GMP	LEDA	LN
2	0.1	7.6	8.0	6.1	88.4	0.2
3	0.8	15.9	17.8	37.9	336	0.4
4	2.1	36.7	41.9	146	988	0.7
5	3.8	125	131	538	2440	3.7
6	6.3	223	236	1300	5170	
7	9.7	374	382	2790	10200	
8	14.1	568	595	5120	18100	
9	19.5	861	887	9400	30300	
10	26.4	1270	1310	15400	48300	
12	44.0	2460	2520	37500	123000	
14	68.3	4400	4470	75700	251000	

Table 1: Performance on random determinants.

n	FP	MOD	PROB	GMP	LEDA	LN
2	0.1	8.2	8.2	6.3	84.6	0.6
3	0.8	21.6	12.8	27.0	232	3.5
4	2.0	36.9	27.8	98.9	699	13.3
5	3.8	119	71.0	221	1100	87.1
6	6.3	234	145	731	2880	
7	9.6	384	203	1110	3590	
8	14.3	586	348	2800	6610	
9	19.4	891	486	3430	10700	
10	26.3	1320	738	6100	18300	
12	43.9	2460	1410	13300	38400	
14	68.1	4460	2450	27200	51800	

Table 2: Performance on small determinants.

n	FP	MOD	PROB	GMP	LEDA	LN
2	0.1	9.3	4.8	6.4	78.7	0.6
3	0.8	19.8	7.4	38.5	284	3.6
4	2.1	39.7	11.9	149	717	13.2
5	3.8	134	25.9	547	1750	86.8
6	6.3	239	41.2	920	4850	
7	9.7	397	58.7	2650	7300	
8	13.9	594	76.2	4720	13800	
9	19.6	894	105	7700	25500	
10	26.3	1330	137	12600	46800	
12	43.7	2550	216	30800	87700	
14	67.9	4460	328	68100	209000	

Table 3: Performance on zero determinants.

obvious improvement, with a reasonably accurate f.p. filter, the penalty of exact arithmetic can be paid only for small determinants (tables 2 and 3).

Some side effects may occur, due to the way we generate matrices. The code of the modular package is freely available (see the CGAL License terms), and anyone can benchmark it on the kind of matrices that he uses. The code will be incorporated in the geometric algorithms library CGAL [11]. It is available via the URL :
`http://www.inria.fr/prisme/personnel/pion/progs/modular/`.

8 Conclusion

RNS systems have been used in number systems because they provide a highly parallelizable technique for multiprecision. As parallel and multi-processor computers are becoming more available, RNS provide an increasingly desirable implementation of multiprecision. This comes in sharp contrast with other multiprecision methods that are not easily parallelizable. Perhaps the main problem with RNS is that comparisons and sign computations seem to require full reconstruction and, therefore, use standard multiprecision arithmetic. We show that one may in fact use only single precision and still perform these operations exactly and efficiently. In some applications, the number of moduli may be large. Our algorithms may be easily implemented in parallel with a speedup depending almost linearly on the number of processors.

As an application, we show how to compute the sign of a determinant. This problem has received considerable attention, yet the fastest techniques are usually iterative and do not parallelize easily. Moreover, they usually only handle single precision inputs. Our techniques are comparable in speed or even faster than other techniques (e.g. [3, 9, 12]), and can easily handle arbitrarily large inputs.

A central problem we plan to explore further is to design algorithms that compute upper bounds on the quantities involved to determine how many moduli should be taken. For determinants, the static bounds we use seem to suffice for applications in computational geometry [21]. They might be overly pessimistic in other areas (such as tolerancing or symbolic algebra) where the nature of the data or algebraic techniques might imply much better bounds. A valid approach we will further study and implement is Newton's incremental method of section 4, provided that we are willing to afford some small probability of error.

Ackowledgments. We thank Christoph Burnikel and Raimund Seidel whose comments improved the contents and presentation of this paper.

References

[1] A. V. Aho and J. E. Hopcroft and J. D. Ullman. *The Design And Analysis Of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] V. Akman and R. Franklin. On the Question "Is $\sum_1^n \sqrt{a_i} \leq L$?". *Bull. EATCS*, 28:16–20, 1986.

[3] 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.

[4] S. Basu, R. Pollack, and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. In *Proc. IEEE Symp. Foundations of Comp. Sci.*, Sante Fe, New Mexico, 1994.

[5] D. Bini, V. Y. Pan. *Polynomial and Matrix Computations. Vol. 1: Fundamental Algorithms*. Birkhäuser, Boston, 1994.

[6] H. Brönnimann, M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 166–173. Also Research Report 3140, INRIA Sophia-Antipolis, 1997.

[7] H. Brönnimann, I. Z. Emiris, V. Y. Pan, S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 174–182.

[8] B. Buchberger, G. E. Collins, and R. Loos, editors. *Computer Algebra: Symbolic and Algebraic Computation*. Springer, Wien, 2nd edition, 1982.

[9] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995. Package available at `http://www.mpi-sb.mpg.de/LEDA/leda.html`

[10] J. F. Canny. An improved sign determination algorithm. In H.F. Mattson, T. Mora, and T.R.N. Rao, editors, *Proc. Intern. Symp. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lect. Notes in Comp. Science*, volume 539 of *LNCS*, pages 108–117, New Orleans, 1991.

[11] CGAL, version 0.9. ESPRIT IV LTR Project No. 21957. Package available at `http://www.cs.ruu.nl/CGAL/`

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

[13] G. E. Collins and R. Loos. Real zeros of polynomials. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 83–94. Springer, Wien, 2nd edition, 1982.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[15] O. Devillers and F. P. Preparata. *A probabilistic analysis of arithmetic filters* Research Report 2971, INRIA Sophia-Antipolis, 1996.

[16] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9(1):67–104, 1990.

[17] I. Z. Emiris. A complete implementation for computing general dimensional convex hulls. *Intern. J. Computational Geom. & Applications*, 1997. To appear. Preliminary version as Tech. Report 2551, INRIA Sophia-Antipolis, France, 1995.

[18] I. Z. Emiris and J. F. Canny. A general approach to removing degeneracies. *SIAM J. Computing*, 24(3):650–664, 1995.

[19] 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.

[20] 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.

[21] S. Fortune, C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. on Graphics*, 1996.

[22] I. M. Gelfand, M. M. Kapranov, and A. V. Zelevinsky. *Discriminants and Resultants.* Birkhäuser, Boston, 1994.

[23] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic *ACM Comput. Surv.* 32(1):5–48, 1991.

[24] G. H. Golub and C. F. van Loan *Matrix computations.* Johns Hopkins University Press, Baltimore, Maryland, 1996.

[25] C. M. Hoffmann. The problem of accuracy and robustness in geometric computation. Report CSD-TR-771, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, 1988.

[26] C. M. Hoffmann. How solid is solid modeling? In *Applied Computational Geometry*, volume 1148 of *LNCS*, pages 1–8. Springer, 1996.

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

[28] C. Y. Hung, B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. *Computers Math. Applic.* 27(4):23–35, 1994.

[29] J. Keyser, S. Krishnan, and D. Manocha. Efficient B-rep generation of low degree sculptured solids using exact arithmetic. Technical Report 40, Dept. Computer Science, Univ. N. Carolina, Chapel Hill, 1996.

[30] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Massachusetts, 1981 and 1997.

[31] M. Lauer. Computing by homomorphic images. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 139–168. Springer, Wien, 2nd edition, 1982.

[32] M. C. Lin, D. Manocha, and J. Canny. Efficient contact determination for dynamic environments. In *Proc. IEEE Conf. Robotics and Automation*, pages 602–608, 1994.

[33] V. Milenkovic Shortest path geometric rounding. Submitted to *Algorithmica, special issue on implementations of geometric algorithms*, S. Fortune, ed., 1997.

[34] B. Mishra. *Algorithmic Algebra.* Springer, New York, 1993.

[35] V. Y. Pan, Y. Yu, and C. Stewart. Algebraic and numerical techniques for the computation of matrix determinants. *Computers & Math. (with Applications)*, 1997, to appear.

[36] V. Y. Pan. Parallel computation of polynomial GCD and some related parallel computations over abstract fields. *Theoretical Computer Science*, 162:2, 173–223, 1996.

[37] A. Rege. A complete and practical algorithm for geometric theorem proving. In *Proc. ACM Symp. on Computational Geometry*, pages 277–286, Vancouver, June 1995.

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

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

[40] M. N. Vrahatis. Solving systems of nonlinear equations using the nonzero value of the topological degree. *ACM Trans. on Math. Software*, 14(4):312–329, 1988.

[41] J. Wiegley, A. Rao, and K. Goldberg. Computing a statistical distribution of stable poses for a polyhedron. In *Proc. 30th Annual Allerton Conf. on Comm. Control and Computing*, Univ.Ill. Urbana-Champaign, 1992.

[42] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7:3–23, 1997.

[43] C. K. Yap. Exact computational geometry and tolerancing metrology. In D. Avis and J. Bose, editors, *Snapshots of Computational and Discrete Geometry, Vol. 3, Tech. Rep. SOCS-94.50.* McGill School of Comp. Sci., 1995.

[44] 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.

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