Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry

Hervé Brönnimann, Christoph Burnikel, Sylvain Pion

Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry[1]

Hervé Brönnimann,[a] Christoph Burnikel,[b] Sylvain Pion[a]

[a] *INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis cedex, France.*
{Herve.Bronnimann,Sylvain.Pion}@sophia.inria.fr

[b] *Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany.*
burnikel@mpi-sb.mpg.de

Abstract

We discuss floating-point filters as a means of restricting the precision needed for arithmetic operations while still computing the exact result. We show that interval techniques can be used to speed up the exact evaluation of geometric predicates and describe an efficient implementation of interval arithmetic that is strongly influenced by the rounding modes of the widely used IEEE 754 standard. Using this approach we engineer an efficient floating-point filter for the computation of the sign of a determinant that works for arbitrary dimensions. We validate our approach experimentally, comparing it with other static, dynamic and semi-static filters.

1 Introduction

Numerical inaccuracy in the evaluation of arithmetic predicates is one of the main obstacles in implementing geometric algorithms robustly [23]. There are numerous approaches to get the problem under control, among which the immediate solution of exact computation stands out because of its generality [22]. In a series of ideas stemming from [13], the exact computation paradigm was refined to an exact predicate paradigm. In this model, the computation is separated into numerical and combinatorial parts, where numerical inaccuracies can only occur in the numerical part, signs and other discrete information is extracted by predicates (such as orientation of three points in the plane) and the combinatorial part is only manipulation of pointers and is free of any numerical problem. To ensure correct behavior it suffices that the predicate be evaluated correctly, which is different from requiring all numerical computation to be exact. Of course, exact computation guarantees exact predicate computation, but other, more efficient approaches have been proposed [1, 3, 4, 8, 21].

For faster yet exact computation, arithmetic filters were proposed in [13, 18]. They make use of the fact that usually only the exact *sign* of some expression is wanted and not its exact *value*. The basic idea of an arithmetic filter is to compute an approximation of the expression and an upper bound on the numerical error. The sign of this approximation is correct provided that the error bound is small enough. Otherwise the computed approximation is useless and hence other methods (more accurate filters or exact

[1] This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL). A preliminary version has appeared in *Proc. 14th ACM Symp. Comput. Geom.* pages 165–174.

computation) have to be used. Filters have shown efficient both in practice [14, 21] and in theory [9].

The work done so far mainly concerns static and semi–static filters where the error bounds, or at least parts of it, are determined at compile time. Purely static filters are restricted to integral, division-free expressions of small bounded depth and require that good upper bounds on the input variables are known in advance, which is sometimes not the case. Semi-static filters remedy most of the problems, but divisions and square roots can only be handled at the price of a significantly reduced quality of the resulting error bounds. Moreover, to apply a semi-static filter to a certain expression the complete structure of this expression has to be preprocessed. If the preprocessing is done by a precompiler, this usually implies some syntactical restrictions to the user program. On the other hand, dynamic filters have none of the mentioned drawbacks, since both the approximation and the error bound are computed at run-time, but they are often considered too inefficient for the use in computational geometry. First advances have been made to incorporate dynamic filters into the LEDA reals [6].

In this paper, we propose to use *interval analysis* [19, 20, 16] for more efficient dynamic filters. The technique is based on carefully engineered interval arithmetic. Each number is represented by an interval which is guaranteed to contain it. Interval arithmetic is very simple to use and yields the most flexible dynamic floating-point filters we know of. Divisions can be handled as well as square roots and hence the technique is not limited to rational expressions. Of course, the intervals grow wider as roundoff errors propagate. With the IEEE 754 standard for floating-point computations [17], the computed intervals are locally optimal in the sense that every single operation results in the smallest possible interval. Consequently, the produced filters have the maximal achievable probability of success. On the other hand, interval arithmetic is still relatively fast. We measure it to be roughly 3-8 times slower than straightforward floating-point evaluation. Our implementation of interval arithmetic is based on the rounding modes of the IEEE 754 standard.

Many geometric predicates boil down to computing the sign of a determinant. Much effort has already been made towards the exact evaluation of signs of determinants, using various specific solutions such as Clarkson's or the lattice method [8, 1, 4], or using general solutions such as exact integer arithmetic [13] or modular arithmetic [3]. For $d \times d$ determinants, the complexities range from $O(d^3 \log d)$ (Clarkson or lattice) to $O(d^4 \log d)$ (modular) or even $O(d^4 \log d \log\log d)$ (exact integer arithmetic) with a potentially large constant in the asymptotic bounds. For all these methods we observe that they are, practically, several orders of magnitude slower than the straightforward, inexact floating-point evaluation, although Clarkson's and the lattice methods are adaptive: their running times are $O(d^3)$ for matrices which are close to orthogonal and worsen gradually the more ill-conditioned the matrix is. By contrast, filters have the same running time for all matrices, only they may fail to report a helpful answer. In this paper we design a new, fast floating-point filter for computing the sign of a determinant based on interval analysis. Its running time is $O(d^3)$ with a small constant and therefore very close to the (not certified) floating point computation. It fails only for matrices which are singular or nearly singular. We also give a simplified filter for small dimensions, which has a weaker probability of success but is very fast. As this version of the filter crucially uses divisions, semi-static or static error computation

is not used.

There are two intrinsic limitations to the use of interval arithmetic. The first limitation is that interval arithmetic may fail to detect that an expression is zero, which often indicates a geometric degeneracy. In such a case the computed interval enclosure necessarily contains zero, but this makes it impossible to deduce the sign of the expression unless the interval is the single point $\{0\}$. Of course, point intervals are only obtained if no rounding error occurred in our computation, and this only happens reliably when the bit length of the input data is small with respect to the machine precision and to the complexity of the computation.

The second limitation is that, although the computed intervals are optimal for every single operation, they can grow arbitrarily wide for a cascaded sequence of operations (of a priori unbounded depth, as is the case in some recursive or iterative algorithms). For the specific problem of evaluating the sign of a determinant, the latter limitation is none: We show how to combine interval arithmetic with a posteriori error computation to a filter whose effectiveness decreases in practice only very slowly with the dimension. This means that our filter can come into effect for matrices of almost arbitrarily large dimensions.

Our paper is organized as follows. In section 2 we lay out a classification of filters into static, semi-static, and dynamic filters and we discuss their usage in precompiled, hand-coded and fully packaged cascaded computation. In section 3 we introduce the basic notions of interval arithmetic and describe our implementation, along with its use in deriving filters for generic expressions. In section 4 we present a new filter for computing the sign of a determinant using a posteriori error analysis and interval arithmetic. In the discussion of section 5, we give an overview of the efficient methods to obtain verified enclosures in intervals, and see why they do not apply to the problems considered here. We introduce a heuristic measure of the effectiveness of interval analysis for a given expression and justify the effectiveness of the approach of section 4. In section 6 the approach is validated experimentally with a new implementation of interval arithmetic that relies on the rounding modes of the IEEE 754 standard. Beside the mentioned applications we also consider other geometric predicates, such as those encountered in geometric optimization or Delaunay sweep algorithms.

2 Arithmetic filters

Generally speaking, a geometric predicate is a mapping from some finite-dimensional Euclidean vector space to a discrete set. In practice, these predicates are only computed by use of $+, -, \cdot, /, \sqrt{\ }$, comparisons $<$, and boolean expressions thereof.[2] We may therefore restrict our attention to computing the signs of expressions. An expression $\mathcal{E} = \mathcal{E}(x_1, \ldots, x_n)$ is a directed acyclic graph where each leaf holds an input variable x_i and each internal node holds an operator in $\{+, -, \cdot, /, \sqrt{\ }\}$ whose arguments are expressions given by the children of that node. The input variable may be assigned a value in a computable subset of the reals (such as a floating-point value).

[2]Divisions and square roots can always be eliminated, but the resulting expressions may be arbitrarily complex (see e.g. [2].)

By exact evaluation, we mean that each node of the expression is evaluated exactly in a bottom-up manner. Of course, numerical errors may occur if we perform the computation using an approximate type (like some fixed precision floating-point arithmetic). In the undesirable case, these numerical errors may affect the sign of expression. By an error bound, we mean a quantity E that upper bounds the numerical error that occurs at the root node of $\mathcal{E}$. This bound may depend on the concrete values assigned to the input variables, or it may be the supremum over a bounded input domain.

Filters are used to evaluate the sign in a robust manner, while being far quicker than the exact evaluation, in most cases. A filter never returns a wrong answer, but may declare that it cannot safely determine the correct answer by returning NO_IDEA.

In our discussion, we will focus on single precision[3] floating-point filters, for which the expression and the error bounds are evaluated in the floating-point domain, because they have a speed comparable to the simple floating-point evaluation. We distinguish mainly three kinds of such filters, described below. All of them compute a numerical approximation $\widehat{\mathcal{E}}$ of $\mathcal{E}$.

Fully static: Upper bounds $|x_i| \leq X_i$ are known for each i, and $\mathcal{E}$ contains only the operations $+, -, \cdot, \sqrt{}$. This makes it possible to precompute a fixed error bound E for the approximation $\widehat{\mathcal{E}}$ which holds for all inputs. For a particular input, the filter fails if $|\widehat{\mathcal{E}}| \leq E$, otherwise the sign of $\mathcal{E}$ is known safely.

Semi-static: No useful upper bounds on the input variables are known, but there is a simple formula $E = E(x_1, \ldots, x_n)$ that gives an error bound for a particular input, even if E is evaluated with single precision. The structure of E is computed at compile-time and E itself is evaluated at run-time. The expression E has a structure very similar to E if $\mathcal{E}$ is division-free. Again, for a particular input the filter fails if and only if $|\widehat{\mathcal{E}}| \leq E$.

Dynamic: The computation of E is carried along with the evaluation of $\mathcal{E}$ and all computations are done completely at run-time. Typically, for each operation of $\mathcal{E}$, a simple rule determines the error bound for the result of that operation based on its operands and on error bounds on them.

Several arithmetic filters can be combined into one by a simple chain. Here we usually start with a cheap filter that has a relatively low probability of success. Only if this filter fails, we apply a second filter that is more expensive to apply but has a higher probability of success. If the second filter fails, we apply a third filter, and so on. Exact computation can be considered the last of filters, which never fails. The cost of the total computation can then be expressed by the costs of the different filters and by their conditional probabilities of success [9].

Static filters are implemented for instance in LN [14], semi-static filters are described in [1, 5], and dynamic filters are given in EXPR [22] and LEDA [6]. Also Shewchuk [21] approximates $\mathcal{E}$ up to first order error terms, then up to second order errors etc., until the sign can be safely determined. This procedure combines a dynamic

[3] Single precision here means a precision of 53 bits, used by IEEE 754 doubles.

filter according to our description with exact computation, since it can reduce the error to zero if needed.

A static filter does the floating-point evaluation of $\mathcal{E}$ plus one extra comparison, whose running time is usually negligible. Hence its running time is nearly the same as for straightforward floating-point evaluation. The cost of a semi-static filter exceeds that of a static filter by the cost of computing the error bound E, which is typically about as much as for the computation of $\mathcal{E}$. Finally, the cost of a dynamic filter is a constant factor times that of the floating-point evaluation.

3 Interval analysis

The major tool used within our filter is *interval analysis*. The use of interval analysis in the context of matrix operations was originally proposed by Moore [19] and further promoted through a research group directed by Kulisch; see [16] for a survey of the available computational methods. In [16], interval analysis is successfully applied to many basic computation tasks in numerical linear and nonlinear algebra. The problem of computing the sign of determinants considered in the present paper seems to have been overlooked so far in the interval analysis community, however.

3.1 Interval arithmetic

Interval arithmetic deals with *intervals* $[x] = [\underline{x}, \overline{x}]$ of real numbers. The basic interval operations are defined essentially as in [16]. Namely, if both operands $[x] = [\underline{x}, \overline{x}]$, $[y] = [\underline{y}, \overline{y}]$ are finite intervals we set

$$
\begin{aligned}
[x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[x] \cdot [y] &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\
[x]/[y] &= \begin{cases} [x] \cdot [1/\overline{y}, 1/\underline{y}] & \text{if } 0 \notin [y], \\ \mathbb{R} & \text{otherwise} \end{cases} \\
[x]^{1/2} &= \begin{cases} [\underline{x}^{1/2}, \overline{x}^{1/2}] & , \quad \text{if } 0 \le \underline{x} \\ \mathbb{R} & , \quad \text{otherwise} \end{cases}
\end{aligned}
$$

As for the definition of the square root, it can in certain specific cases be modified as follows: $[x]^{1/2} = [0, \overline{x}^{1/2}]$ if $\underline{x} < 0 \le \overline{x}$. This is only safe, however, when the argument of the square root can be shown mathematically to be nonnegative. In that case, the modified definition gives a more meaningful result even though the lower bound of the argument has been rendered negative by roundoff error propagation. This happens in particular if interval arithmetic is used only inside some well-known expression like a geometric predicate. For a general interval arithmetic number type however, in case the interval contains negative values, we have no other safe decision than to say that the expression is ill-formed, hence the definition above.

Since the computed intervals $[x]$ in general have bounds $\underline{x}$, $\overline{x}$ that are not contained in the given finite set $\mathbb{F}$ of floating-point numbers, we compute in each arithmetic step

the smallest interval $\diamond[x] = [\bigtriangledown\underline{x}, \bigtriangleup\overline{x}]$ that encloses $[x]$ such that $\bigtriangledown\underline{x}$ and $\bigtriangleup\overline{x}$ are contained in $\mathbb{F}$. This means that $\bigtriangledown\underline{x}$ (respectively $\bigtriangleup\overline{x}$) is the numbers in $\mathbb{F}$ next to $\underline{x}$ and $\overline{x}$ when rounding downwards (respectively upwards). Then the approximate floating-point interval operations are given by

$$\begin{aligned}
[x] \oplus [y] &= \diamond([x] + [y]) \\
[x] \ominus [y] &= \diamond([x] - [y]) \\
[x] \odot [y] &= \diamond([x] \cdot [y]) \\
[x] \oslash [y] &= \diamond([x]/[y]).
\end{aligned}$$

This notation is adapted to interval functions like $f([x]) = (e^{[x]} - 1)[x]$ by writing $f_\diamond([x])$ for the smallest floating-point interval containing $f([x])$. For the rest of the paper we assume that interval expressions are evaluated using the approximate interval operations $\oplus, \ominus, \odot, \oslash$ and $\text{sqrt}_\diamond([x]) = \diamond([x]^{1/2})$.

3.2 Engineering floating-point filters with interval arithmetic

Let $\mathcal{E}$ be an arithmetic expression over the operations $\{+, -, \cdot, /, \sqrt{\ }\}$. If the expression is applied to real numbers $x_1, \ldots, x_n$, we denote the resulting interval enclosure by $[\mathcal{E}]$ for clarity. In our application, $\mathcal{E}$ is the expression used to evaluate a geometric predicate, with exact input (i.e. points, not intervals). For instance, the orientation of three points, or the intersection test of two segments. We discuss more predicates in section 6. The task is to certify the sign of this expression, returning NO_IDEA in as few cases as possible.

In fact, our interval methods can be applied when the input have interval values. These intervals may arise from uncertainty in the input (e.g., when the input is subject to imprecise measurement) as well as from approximate intermediate calculations. But in the case of geometric predicates, the input is given exactly. This is because we want to use interval arithmetic as a preliminary stage for *exact* computation. Note that when the input is an interval, the predicate does not have a uniquely defined output value. For interval–type input matrices this means that the determinant does not necessarily have a unique sign. This fact cannot be altered by later using exact arithmetic. In that case, we need a completely different, "fuzzy" model of computation that deals with multiple predicate values. We do not discuss such a model here (see e.g. [10]).

The naive filter implementation consists of evaluating $[\mathcal{E}]$ for a given input. When evaluating the sign of $[\mathcal{E}]$, if $0 \in [\mathcal{E}]$ we return NO_IDEA. Otherwise the sign can be safely inferred. (Note that if an interval is $\mathbb{R}$ in the intermediate computation we can cut short and return NO_IDEA without further ado.) This approach has been implemented in CGAL by the last author.[4] Experience shows that it is very efficient (somewhat slower than a semi-static filter, and about 3 to 8 times slower than inexact floating-point computation) and that it rarely fails on non-degenerate instances, even those for which a semi-static filter fails.

Determinants are particular, division-free expressions which can be evaluated by developing along a row or a column. The resulting expression is only efficient up to

[4] `http://www.cs.uu.nl/CGAL/`

dimension 5 [13], however, beyond which other methods (such as LU decomposition) are more efficient. In the next section, we develop filters for determinants and LU decomposition.

4 Computing the sign of a determinant

Let $\mathbb{F}$ be a set of *fixed precision* floating-point numbers. The problem that we consider in this section is the following: given a matrix $A \in \mathbb{F}^{d,d}$ over $\mathbb{F}$, safely compute the exact sign of $\det(A)$. This is an important problem in computational geometry since many geometric predicates are expressible by determinants.

In this section we design fast *floating-point filters*, with running time $O(d^3)$ comparable to floating-point computation, without imposing any restriction on the input. The filters are applicable also if the input is not exactly representable in $\mathbb{F}^{d,d}$. The filters fail only for matrices that are singular or nearly singular. Our first filter is of use mostly for small dimensions. Our second filter, while being less efficient by a constant factor, almost always succeeds even for dimensions up to $d = 800$.

4.1 Naive interval arithmetic

Our first algorithm is a straightforward use of interval arithmetic. One of the standard methods in numerical analysis to compute a determinant $\det(A)$ uses the LU decomposition $P \cdot A = L \cdot U$ where P is a permutation matrix, L is lower triangular and U is upper triangular. Of course, using fixed precision arithmetic we will get only an approximate decomposition $P \cdot A \approx L \cdot U$. The determinant of P is ± 1 and can be computed without rounding error, and $\det(L)$ is 1 since the diagonal elements of L are all equal to 1 by construction (their corresponding intervals are points $\{1\}$). So the product of the diagonal elements $u_{i,i}$ of U gives an approximation of $\pm\det(A)$. We therefore compute an interval enclosure of these approximations.

Algorithm 1 *(returns the sign of $det(A)$, if successful)*

1. *Compute $P \cdot A = [L] \cdot [U]$ with partial pivoting as in [15], except that all operations are performed in interval arithmetic instead of floating-point.*
2. *If one of the intervals $[u_{i,i}]$ on the diagonal of $[U]$ contains zero, return NO_IDEA. Otherwise, return $sign(det(P)) \cdot \prod_i sign([u_{i,i}])$ where $sign([u_{i,i}])$ is the sign of any number in $[u_{i,i}]$.*

Remark 1. Step 1 can always be completed, since the division by a "pivot" interval containing 0 is a regular interval operation that results in $[-\infty, \infty]$. In that case, however, it is more relevant to terminate early, and simply return NO_IDEA.
Remark 2. The lower triangular part $[L]$ does not have to be computed explicitly, since it is not needed in the sign calculation and its determinant is known to be 1 exactly.

We measured that algorithm 1 is only at most 2.6 times slower than with naive floating-point calculation. This running time only depends on our implementation of

interval arithmetic and does not depend on the entries (random or degenerate) nor on the dimension. Indeed,

Lemma 1 *Algorithm 1 takes at most $d^3/3 + O(d^2)$ interval operations (1 operation = 1 addition + 1 multiplication).*

See section 6 for more detailed measurements.

The probability of success, however, crucially depends on the dimension. Algorithm 1 is best applied to matrices of small dimension. Indeed, for large dimensions, the complexity of step 1 means that the intervals grow very large and therefore the probability of finding a pivot which does not contain 0 decreases (see section 5 for a heuristic justification of this). In practice, the algorithm is useless for large matrices. For large matrices with a special structure, like certain sparse matrices, diagonally dominant matrices and block matrices of small block size, however, it might still be useful.

Note that algorithm 1 cannot be enhanced by the standard trick of multiplying A with an approximate inverse A_{inv} to make it diagonally dominant. As already mentioned, it is as hard to compute the sign of $\det(A_{inv})$ as that of $\det(A)$. Another possible preconditioning matrix is $L_{inv}P$ where L_{inv} is an approximate inverse of L whose determinant is known to be 1. In fact, the resulting interval enclosure $[\tilde{U}]$ of $L_{inv}PA$ is nearly upper triangular. Unfortunately, the following Gaussian elimination that makes $[\tilde{U}]$ truly upper triangular incurs intervals that are not smaller, or even larger than those obtained by doing Gaussian elimination directly on A. We do not know how to find a preconditioning matrix whose determinant is both easy to compute and leads to reduced interval sizes.

To be complete, observe that this approach not only gives us the sign but an interval enclosure of the value of $\det(A)$, by computing $\mathrm{sign}(\det(P)) \cdot \prod_i [u_{i,i}]$. Again, we do not need to compute the diagonal elements $[l_{i,i}]$ of $[L]$, which are all point intervals equal to 1.

4.2 A posteriori method

For well-conditioned matrices the floating-point approximation of L and U is usually quite reliable, although it cannot always be certified by direct application of interval arithmetic as we mentioned in the previous section. In the rest of the section we shall investigate under what circumstances we can conclude that $\det(P \cdot A)$ has the same sign as $\prod_i \mathrm{sign}(u_{i,i})$. Since $P \cdot A \approx L \cdot U$ we expect that $A^{-1} \approx U^{-1}L^{-1}P$. Because we cannot evaluate U^{-1} and L^{-1} exactly, we invert U and L numerically to matrices $U_{inv} \approx U^{-1}$ and $L_{inv} \approx L^{-1}$ which gives the approximate inverse

$$B := U_{inv} L_{inv} P$$

of A. Note that by the triangular structure of L and U, L_{inv} still has all diagonal elements equal to 1 and the diagonal elements of U_{inv} are $1 \oslash u_{i,i}$. This shows that $\det(B)$ has the same sign as $\det(U)\det(P)$. Since $\det(A) = \det(B)^{-1}\det(B \cdot A)$ it follows that $\det(A)$ has the same sign as $\det(B)$ if we can show that $\det(B \cdot A) > 0$. Here we use the following lemma, valid for any norm such that $\|Fx\| \leq \|F\| \cdot \|x\|$ (usually called a matrix norm).

Lemma 2 *Let $\|F\| < 1$ for some fixed matrix norm. Then $\|(I+F)^{-1}\| \leq \frac{1}{1-\|F\|}$ and det$(I+F) > 0$.*

Proof. $I + F$ is a non-singular matrix since for a vector $x \neq 0$ we have

$$\|(I+F)x\| = \|x + Fx\| \geq \|x\| - \|Fx\| \geq \|x\| - \|F\|\|x\| = (1 - \|F\|)\|x\| > 0.$$

This also shows that $\|(I+F)^{-1}\| \leq \frac{1}{1-\|F\|}$. Consider the mapping D from $[0,1]$ to $\mathbb{R}$ such that $D(t) = \det(I + t \cdot F)$. D is continuous and $D(0) = 1$. Since $I + t \cdot F$ is non-singular by the argumentation above, $D(t)$ is always nonzero for t in $[0,1]$. By continuity of D, this means that $D(1) = \det(I+F) > 0$. □

Using the lemma, checking that $\det(B \cdot A) > 0$ can be done by checking whether the defect matrix $E = I - B \cdot A$ has norm less than 1 for some matrix norm of our choice. To this end, assume we have an interval enclosure $[I - B \cdot A] = ([e_{i,j}])_{i,j}$ of $I - B \cdot A$. Using the norm $\|\cdot\|_\infty$ we have to test whether the sum of the maximal possible absolute values of $e_{i,j}$ is smaller than 1 in every row. We denote the maximal row sum in this calculation by $\|[I - B \cdot A]\|_\infty$. The description of our algorithm for the computation of sign(det(A)) is as follows. Note that the algorithm either returns $+1$, -1, or the string NO_IDEA (if the filter fails).

Algorithm 2 *(returns the sign of det(A), if successful):*

1. *Compute a numerical decomposition $P \cdot A \approx L \cdot U$ with partial pivoting as in [15]. If this is not possible for numerical reasons, return NO_IDEA.*
2. *Compute numerical inverses $U_{inv} \approx U^{-1}$ and $L_{inv} \approx L^{-1}$. In case of exponent overflow in the floating-point computation, return NO_IDEA.*
3. *Compute $\|[I - BA]\|_\infty$ where $[I - B \cdot A]$ is computed as $I - U_{inv}(L_{inv}(PA))$ by using interval arithmetic.*
4. *If $\|[I - BA]\|_\infty < 1$, return sign(det(P)) $\prod_i$ sign$(u_{i,i})$. Otherwise return NO_IDEA.*

Remark 3. The computation of the numerical LU decomposition in Step 1 can fail if all elements of a pivot column are zero. In practice this occurs if A is singular or very nearly singular.

Remark 4. If Step 1 was correctly completed, Step 2 can only fail because of exponent overflow.

Remark 5. There seems to be no easy way to improve the approximate inverse B if the quality does not turn out to be sufficient in Step 3. The reason is that B does not only have to be a good approximate inverse of A, but also sign(det(B)) has to be computable exactly.

Lemma 3 *Algorithm 2 takes at most $d^3 + O(d^2)$ floating-point operations and $d^3 + O(d^2)$ interval operations (1 operation = 1 addition + 1 multiplication).*

Proof. The computation of the LU decomposition takes $d^3/3 + O(d^2)$ operations, and so does each inversion of a triangular matrix. This accounts for the floating-point operations. Now we count the interval operations. We have to compute two products of a full matrix with a triangular matrix, of which each one requires $d^3/2 + O(d^2)$ operations. All other computations require only $O(d^2)$ operations. □

Let us now give a rough estimate for the running time of algorithm 2 with respect to the unfiltered evaluation of $\text{sign}(\det(A)) \approx \text{sign}(\det(P))\prod_i \text{sign}(u_{i,i})$ (which we call the "floating-point algorithm"). Hammer et al. [16] report that interval computations roughly take double time than ordinary floating-point calculations. In our own C++ implementation, this (clearly optimal) overhead was for matrix computations only nearly achieved; an overhead factor of about 3 to 4 is realistic, though. Our measurements showed that algorithm 2 takes in fact 9 to 12 times longer than Algorithm 1. If algorithm 1 returns a result, it always outperforms algorithm 2. On the other hand, algorithm 2 has a higher probability of success since the computed intervals are smaller. See section 6 for detailed measurements.

For the sake of completeness, we sketch how to compute not only the sign of the determinant but also a verified enclosure for the *value* of the determinant. We need a slightly stronger version of Lemma 2 that uses the Euclidean matrix norm.

Lemma 4 *For $\|F\|_2 < 1$, we have $|det(I + F) - 1| < (1 + \|F\|_2)^d - 1$.*

Proof. Let $\sigma_i, i = 1, \ldots, d$ be the (nonnegative) singular values of the matrix $I + F$. As we already know by Lemma 2, $\det(I + F) > 0$ and hence $\det(I + F) = \prod_i \sigma_i$. Now recall that the singular values of I (which are all 1) and the σ_i can differ by at most $\|F\|_2$, see Corollary 8.6.2 of [15]. This implies that

$$|\det(I + F) - 1| = |\prod_i \sigma_i - 1| \leq (1 + \|F\|_2)^d - 1. \qquad \square$$

We can apply Lemma 4 in the context of Algorithm 2 with $F = B \cdot A - I$. If the coefficients of F are all bounded by φ, we can bound the $\|F\|_2$ by $d\varphi$ and this bound is hopefully much smaller than 1. Hence in this case by Lemma 4

$$\epsilon = |\det(B \cdot A) - 1| = |\det(I + F) - 1| \leq (1 + d\varphi)^d - 1 \approx d^2\varphi \ll 1.$$

Assume we know a bound $\eta > 0$ such that $\epsilon \leq \eta$, then we have an enclosure of $[y] = [1 - \eta, 1 + \eta]$ for $\det(B \cdot A)$. By evaluating $[x] = [\det(B)] = \prod_i [u_{i,i}]^{-1}$ using interval arithmetic we obtain the desired enclosure $[y]/[x]$ of $\det(A)$. One possible value of η is readily obtained by setting $\eta = (1 + d\varphi)^d - 1$ as in Lemma 4, but it is very not very accurate. To derive a better bound η, one can either use interval arithmetic again, or else one can use classical numerical analysis.

5 Discussion

5.1 A heuristic measure of quality

It turns out that the interval evaluation of expressions is not always effective, depending on the particular structure of the expression. This notion of effectiveness is related to

the size of the resulting intervals, not to the time necessary to evaluate the interval expression. This is because the interval evaluation incurs only a constant overhead over the usual floating-point approximation. [5] Important types of expressions that are well suited for interval evaluation are *dot products* or *inner products* of vectors and the derived operations of matrix-matrix product and matrix-vector product. The following *interval degree Ideg*$(\mathcal{E}) \in \mathbb{Z}$ is a heuristic, asymptotic measure for the average relative width of the interval $[\mathcal{E}]$ and hence for the quality of the interval evaluation of $\mathcal{E}$. For an expression $\mathcal{E}$ consisting of a single input number z we set Ideg$(\mathcal{E}) = 0$ and inductively, if $\mathcal{E}$ is computed from expressions $\mathcal{X}$, $\mathcal{Y}$ by the operations $\{+, -, *, /, \sqrt{\ }\}$ we set

$$\begin{array}{rcl} \text{Ideg}(\mathcal{X} + \mathcal{Y}) & = & \max\{\text{Ideg}(\mathcal{X}), \text{Ideg}(\mathcal{Y})\} \\ \text{Ideg}(\mathcal{X} - \mathcal{Y}) & = & \max\{\text{Ideg}(\mathcal{X}), \text{Ideg}(\mathcal{Y})\} \\ \text{Ideg}(\mathcal{X} \cdot \mathcal{Y}) & = & 1 + \max\{\text{Ideg}(\mathcal{X}), \text{Ideg}(\mathcal{Y})\} \\ \text{Ideg}(\mathcal{X}/\mathcal{Y}) & = & 1 + \max\{\text{Ideg}(\mathcal{X}), \text{Ideg}(\mathcal{Y})\} \\ \text{Ideg}(\mathcal{X}^{1/2}) & = & \text{Ideg}(\mathcal{X}). \end{array}$$

In this notation, the mentioned products all have interval degree 1. On the other hand, many of the basic operations in linear algebra have larger interval degree. For example, the degrees of computing $\det(A)$, of computing decompositions $P \cdot A = L \cdot U$, and of solving triangular systems for a d dimensional matrix are all $\Theta(d)$. Typically, the computed intervals are useless if Ideg$(\mathcal{E})$ has the same order of magnitude than the used mantissa length p of the floating-point numbers and are very useful if Ideg$(\mathcal{E})$ is a small constant. As an example for the calculation of interval degree we prove the following lemma.

Lemma 5 *The interval solution of a triangular system $T \cdot [x] = b$ with exact input data has interval degree d, where d is the dimension of T and b.*

Proof. Without loss of generality, we consider forward substitution with a lower triangular matrix $T = (t_{i,j})_{i,j}$. In the basic step $d = 0$ we have $x_1 = b_1/t_{1,1}$ which has interval degree 1. For $d > 1$ assume by induction that $x_1, \ldots, x_{d-1}$ have interval degree at most d and Ideg$(x_{d-1}) = d - 1$. From the formula

$$x_d = b_d/t_{d,d} - \sum_{j=1}^{d-1} x_j(t_{d,j}/t_{d,d})$$

we see that since Ideg$(t_{d,j}/t_{d,d}) = 1$, each product $x_j(t_{d,j}/t_{d,d})$ has interval degree at most d. Because the latter term has interval degree d for $j = d$, the whole sum has interval degree d. Our statement follows by induction. □

Experiments with a randomly chosen matrix A show that the interval solution of a system $A[x] = b$ using a LU decomposition of A in fact incurs an uncertainty in the last $\Theta(d)$ places of the interval bounds, even if A is tridiagonal. Another simple lemma is that Gauss elimination is of interval degree $2d$.

[5] This overhead depends on the particular platform given by hardware architecture, programming language and compiler.

Given that many of the most important computations in linear algebra have non-constant interval degree, we can now better understand the efficiency of algorithm 2. Algorithm 2 is applicable even for very large dimensions; it was successfully tested up to $d = 800$. Note that $I - U_{inv}(L_{inv}(PA))$ computed in step 3 of this algorithm is a computation of constant interval degree 3, because U_{inv} and L_{inv} are numerical (and not interval) approximations of U^{-1} and L^{-1}. Thus the a posteriori certification of the quality of the floating point solution has a much smaller interval degree, while the quality of that solution depends only on the condition number of the matrix and involves no catastrophic interval error propagation.

Let us stress again that our notion of interval degree gives a *purely heuristic and asymptotic* estimate of the usefulness of interval arithmetic. Our point is that interval arithmetic can be useful for expressions with small bounded interval degree or else if the problem has small dimension. Nevertheless, our interval degree is logarithmically related to the index of [5] (if we do not count additions), and this index gives a bound on the relative error of a computation. For problems of large interval degree in large dimensions, interval arithmetic can still be useful if the input data has an appropriate (sparse) structure.

5.2 General techniques

As we just said, many of the most important computations in linear algebra have non-constant interval degree. How can we save the interval method? There are several powerful complementary approaches of which we cite the following three.

1. *A posteriori error analysis:* Compute first a candidate for the solution by conventional floating-point arithmetic and then use this candidate to compute an interval enclosure for the solution by interval arithmetic.

2. *Preconditioning:* First bring the problem in a form that is better suited for interval computation, by applying an appropriate preconditioning transformation.

3. *Fixed point theorems:* Reformulate the problem such that the solution is expressed as the limit of an iteration process $x \leftarrow f(x)$ where $f(x)$ has small interval degree in the unknown x. Try to find an interval I such that $f(I) \subset I$ can be shown (again by interval arithmetic). Now use an appropriate fixed point theorem like the classical one by Banach or the more sophisticated by Brouwer [20] and try to prove that the iteration converges to a solution contained in I.

These methods work well for many of the basic problems in numerical mathematics, including solutions of linear and nonlinear systems, global optimization and automatic differentiation [16]. Applying the methods to the computation of a determinant is not straightforward, however. We have shown in section 4.2 how to do it for the a posteriori method. The standard way of preconditioning in the case of a linear problem is to first multiply the matrix A with an approximate inverse A_{inv} of A and then apply e.g. Gaussian elimination to the resulting interval matrix. This is not helpful in our case because it is no easier to compute the determinant sign for the approximate inverse of A than for A itself. Finally, the fixed point method does not seem to work: If

A is a symmetric matrix, then we could compute the determinant as the product of the eigenvalues of A. Given an approximate eigenvector x' and an approximate eigenvalue λ' of A, we could further use a Newton-Raphson iteration that converges to the desired solution of $(A - \lambda I)x = 0$ as in [16]. However, in the case of a non-symmetric matrix A, only the absolute value of $\det(A)$ is expressible by the product of the singular values of A. Hence we cannot handle the general case in this way.

There are also certain restrictions to the interval method. First of all, interval arithmetic almost always fails if the matrix is singular or nearly singular. In such cases the user should apply one of the exact algorithms. Second, the interval method of algorithm 2 can in general not benefit (much) from sparsity in the matrix A. This is because it computes an approximate inverse of A, which is usually a dense matrix. On the other hand, algorithm 1 can often profit from sparsity, but it is essentially restricted to small dimensions. It is an open problem how to design a more efficient filter for computing the determinant sign of general large sparse matrices.

6 Experimental Validation

We experiment with our filter in a variety of settings encountered by geometric algorithms.

Implementation of basic interval operations. We implemented interval operations by C++ operators, using the rounding mode of the IEEE Standard 754. Since we maintain here the upper interval bound and the *negative* of the lower interval bound we can always round upwards, except for the square root operation. Let two intervals $[x] = [-\underline{x}, \overline{x}]$ and $[y] = [-\underline{y}, \overline{y}]$ be given. For addition we compute the smallest interval $[z] = [-\underline{z}, \overline{z}]$ such that $[x] + [y] \subset [z]$ by setting $\underline{z} = \triangle(\underline{x} + \underline{y})$ and $\overline{z} = \triangle(\overline{x} + \overline{y})$. Subtraction is implemented very similarly. Multiplications and divisions are slightly more complex because they require some case distinctions for the various possible signs of $\underline{x}, \overline{x}, \underline{y}, \overline{y}$. For the square root operation we take the square root of the lower bound, rounding downwards, and the square root of the upper bound, rounding upwards.

We stress that the advantage of taking the negative of the lower interval bounds is that within a sequence of rational operations the rounding mode never has to be adjusted, if once set to $+\infty$. For completeness, we also implemented a safer (but less efficient) version where the rounding mode is set and reset *within* each operator call, which we call below our "protected" implementation.

To implement interval arithmetic we derived two C++ classes with overloaded operators $+, -, \cdot, /, \sqrt{}$, an efficient version (`Interval_nt_advanced`, "Advanced") where the user is responsible to have upwards rounding active throughout the interval computations, and a protected version (`Interval_nt`, "Protected"). The implementation is available in CGAL and independently at `http://www-sop.inria.fr/prisme/personnel/pion/`. Although it is only possible to evaluate the efficiency of our implementation in a concrete *sequence* of interval operations, we measure `Interval_nt_advanced` to be roughly 2 to 8 times slower than floating point computation, without counting the rounding mode selection.

Isolated geometric predicates. This section presents some benchmarks on isolated well known low-dimensional predicates, to show the overhead caused by the use of various filters, compared to the pure floating-point computation.

For comparison, we evaluate the semi-static filter given in [5]. Furthermore we also tested two other, completely different dynamic filters that are improved versions of the filter used in the number type `leda_real` [6], which we call "AbsFilter" and "RelFilter". These filters use standard error analysis to dynamically propagate absolute and relative errors for each arithmetic operation.

The considered predicates are the standard orientation and in-sphere predicates, and a specific in-circle predicate for the Voronoi diagram of line segments that asks whether a point lies in the circle circumscribed to two lines and a point site. The latter predicate involves three square root operations but no divisions.
Points are represented in Cartesian coordinates, and lines by their equations. We used the following expressions:

$$\text{orientation2}(p_1,p_2,p_3) = \begin{vmatrix} x_2-x_1 & x_3-x_1 \\ y_2-y_1 & y_3-y_1 \end{vmatrix},$$

$$\text{orientation3}(p_1,p_2,p_3,p_4) = \begin{vmatrix} x_2-x_1 & x_3-x_1 & x_4-x_1 \\ y_2-y_1 & y_3-y_1 & y_4-y_1 \\ z_2-z_1 & z_3-z_1 & z_4-z_1 \end{vmatrix}$$

$$\text{insphere3}(p_1,p_2,p_3,p_4) = \begin{vmatrix} x_2-x_1 & y_2-y_1 & z_2-z_1 & (x_2^2+y_2^2+z_2^2)-(x_1^2+y_1^2+z_1^2) \\ x_3-x_1 & y_3-y_1 & z_3-z_1 & (x_3^2+y_3^2+z_3^2)-(x_1^2+y_1^2+z_1^2) \\ x_4-x_1 & y_4-y_1 & z_4-z_1 & (x_4^2+y_4^2+z_4^2)-(x_1^2+y_1^2+z_1^2) \\ x_5-x_1 & y_5-y_1 & z_5-z_1 & (x_5^2+y_5^2+z_5^2)-(x_1^2+y_1^2+z_1^2) \end{vmatrix}$$

where the 3×3 and 4×4 determinants are computed by developing along the last row recursively and with dynamic programming to factor in the common 2×2 subdeterminants. The implementation of the in-circle predicate follows from [7], Section 2.3.1.3. The benchmarks of Tables 1 were made on a Ultra Sparc IIi, 333 MHz, with the GNU C++ compiler version 2.95 and using the flag -O2.

As our experiments show, the variant "Advanced" is always the fastest. Hence in all other experiments, we will use only this filter variant. Table 1 documents only differences in the running times, without accounting for the subsequent computations in case of failure.

Since we are using random entries, the probabilities of success of these filters are very high. In a more real situation, the filters may have different probabilities of success. For a fair comparison, one needs to evaluate them within some geometric algorithm. In both sections below, we leave aside the running time (also called *efficiency*) to concentrate on the success rates (also referred to as *efficacy*) of our filters.

Computing determinants.

We investigate the efficacy of Algorithms 1 and 2 on some nearly singular matrices. Our test matrices have all floating-point entries close to 1 but with a random perturbation of order 2^{-p}, i.e., after the pth bit. The efficacy of Algorithm 1 was measured as

Filter	Insphere 3D	Orientation 3D	Orientation 2D	Incircle 2D (llp_p)
Pure f.-p.	1.00	1.00	1.00	1.00
Semi-static	2.64	1.99	1.68	1.82
Advanced	8.30	3.27	2.56	5.90
Protected	41.3	19.7	10.9	19.0
AbsFilter	44.5	12.5	6.24	21.4
RelFilter	44.2	19.8	9.78	21.7

Table 1: Running time overhead caused by the use of various filters, compared to pure floating-point computation (without error checking and correctness guarantee). The time taken by subsequent computations in case of filter failure is not taken into account.

the maximal relative error of a pivot element, averaged over a large number of random choices for the matrix elements. Here the relative error of a pivot element approximated by the interval $[x, y]$ not containing 0 is defined as $(y - x)/|x + y|$. Note that this value is always smaller than 1 but can be arbitrarily close to 1. If $0 \in [x, y]$ we set the error to 2.[6] The results for various values of the dimension and the perturbation parameter p are shown in Figure 2. Likewise, we measure the efficacy of Algorithm 2 in terms of the quantity $\delta = \|[I - BA]\|_\infty$, if this norm is smaller than 1 and otherwise we set $\delta = 2$. The average values of δ for Algorithm 2 are shown in Figure 3.

n	$p = 1$	$p = 6$	$p = 12$	$p = 18$	$p = 24$	$p = 32$	$p = 40$
6	1.7e-13	5.2e-12	2.1e-10	9.8e-08	9.3e-06	0.00085	0.063
12	2e-11	1.1e-09	3.4e-08	1.4e-06	0.00014	0.027	> 1
18	4.8e-10	2e-07	1.3e-06	8e-05	0.0053	0.67	> 1
24	4.6e-08	1.6e-06	0.00016	0.016	0.23	> 1	> 1
32	5.4e-06	0.00013	0.011	0.51	> 1	> 1	> 1
40	0.0015	0.061	0.93	> 1	> 1	> 1	> 1
48	0.15	> 1	> 1	> 1	> 1	> 1	> 1

Table 2: The average maximal relative error of a pivot for the naive method

Another meaningful parameter to investigate the quality of the filter is the minimal value $k = k(n)$ of the parameter p such that the failure rate gets above 50%. In table 4 we display this number k, first for Algorithm 1 and then for Algorithm 2. A dashed entry means "fails always." We conclude from Tables 2, 3 and 4 that Algorithm 2 is much less sensitive to the dimension than Algorithm 1. In particular, Algorithm 2 can handle fairly difficult input matrices of large sizes. On the other hand, Algorithm 1 is pretty useless for dimensions greater than the mantissa length. Empirically, we find that the number $k(n)$ decreases linearly with n for Algorithm 1 and decreases

[6] It would be undesirable to set the error to ∞ because otherwise a single pivot interval containing 0 would make the average error infinite. If this consistently happens, though, any relative error higher than 1 means failure for the sign determination, and the value 2 fulfills this purpose as well as ∞.

n	$p=1$	$p=6$	$p=12$	$p=18$	$p=24$	$p=32$	$p=40$
6	2.6e-13	7.7e-12	5.4e-10	4.5e-08	2.6e-06	0.00056	0.097
12	4.3e-12	5.8e-11	2.4e-09	2.7e-07	1.9e-05	0.0031	0.47
18	7.9e-12	1.3e-10	6.8e-09	1.6e-06	5.5e-05	0.007	0.98
24	7.2e-12	4.4e-10	1.2e-08	1.5e-06	6.4e-05	0.017	> 1
32	1.2e-10	6.8e-10	1e-07	1e-06	9.2e-05	0.02	> 1
40	2.7e-11	7.9e-10	3.3e-08	2.5e-06	0.0002	0.026	> 1
48	3.8e-11	8.3e-10	8.4e-08	4.2e-06	0.00092	0.1	> 1

Table 3: The average defect norm for the a posteriori method

method, n	6	8	10	12	14	16	20	24	28	32	40	48	56
Algorithm 1	46	44	42	40	37	35	32	28	24	21	13	5	–
Algorithm 2	45	44	43	42	42	41	40	39	39	39	38	38	36

Table 4: The minimal value of p for which the interval filters fail for at least 50% of the cases.

sub-linearly with n for Algorithm 2. Note that we could not determine the maximum dimension for which Algorithm 2 still works, simply because this value is so large that the matrix inversion is not practically feasible anymore. The maximal value that was (successfully) tested was $n = 800$.

And in a sweep algorithm for Voronoi. We have incorporated a floating-point filter into our implementation of the sweep algorithm for building Voronoi diagrams [11, 12]. The predicates involve orientation tests, comparing the ordinates of a point and of an intersection of two parabola, and comparing between elements of a set of abscissae of points or maximum abscissae of circumscribed circles. The latter is the more demanding predicate as it uses square roots and has Ideg 4, but its exact computation with integers would require 20-fold precision. The predicates are illustrated in Figure 1 and the expressions used to compute them are given below (beside the orientation test given before). The first is a well known expression to compute the center and radius of a circumscribed circle. First let D_x, D_y, D_z and D be the cofactors of x, y, x^2+y^2 and 1 in the expansion with respect to the last column of

$$\text{Cocyclicity}(p_1,p_2,p_3,p) = \begin{vmatrix} x_1 & x_2 & x_3 & x \\ y_1 & y_2 & y_3 & y \\ x_1^2+y_1^2 & x_2^2+y_2^2 & x_3^2+y_3^2 & x^2+y^2 \\ 1 & 1 & 1 & 1 \end{vmatrix}.$$

Note that the equation of the circumscribed circle is $\text{Cocyclicity}(p_1,p_2,p_3,p) = 0$. We compute D_x, D_y, D_z and D by the following expressions:

$$D_x = -\begin{vmatrix} y_2-y_1 & (x_2+x_1)(x_2-x_1)+(y_2+y_1)(y_2-y_1) \\ y_3-y_1 & (x_3+x_1)(x_3-x_1)+(y_3+y_1)(y_3-y_1) \end{vmatrix},$$

$$D_y = \begin{vmatrix} x_2 - x_1 & (x_2 + x_1)(x_2 - x_1) + (y_2 + y_1)(y_2 - y_1) \\ x_3 - x_1 & (x_3 + x_1)(x_3 - x_1) + (y_3 + y_1)(y_3 - y_1) \end{vmatrix},$$

$$D_z = -\text{orientation2}(p_1, p_2, p_3),$$

$$D = \begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 \\ x_2 - x_1 & y_2 - y_1 & (x_2 + x_1)(x_2 - x_1) + (y_2 + y_1)(y_2 - y_1) \\ x_3 - x_1 & y_3 - y_1 & (x_3 + x_1)(x_3 - x_1) + (y_3 + y_1)(y_3 - y_1) \end{vmatrix}.$$

The value of D is computed again by dynamic programming, as in orientation3. This yields

$$x_c = \frac{D_x}{2D_z}, \qquad {r_c}^2 = \frac{D}{D_z} - \frac{D_x^2 + D_y^2}{4D_z^2}$$

Hence the quantity used by the algorithm is computed by

$$\text{x\_maximum}(p_1, p_2, p_3) = x_c + \sqrt{{r_c}^2}$$

The second formula we need is the intersection of two parabolas with foci p_1 and p_2 and common director of equation $x = x_l$. Note that y_intersect depends on the order of the arguments, as there are two intersections; the other intersection is obtained by switching the argument p_1 and p_2. We compute it by the following method:

$$a = x_2 - x_1, \qquad b = -2y_1(x_2 - x_l) + 2y_2(x_1 - x_l),$$

$$c = (x_1 - x_2)(x_1 x_2 - (x_1 + x_2)x_l),$$

$$\text{y\_intersect}(p_1, p_2, x_l) = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

The entire algorithm is then evaluated with interval arithmetic as explained in section 3.2. Note that we actually compute once and for all an interval enclosure of the intermediate various quantities, and not each time they appear in a predicate.

We ran the program on a variety of configurations. The first configuration is random; the second is a perturbed grid and the third is a perturbed circle; finally, we ran the program on a degenerate circle. All numbers were generated between $\frac{1}{2}$ and 1 with 53 bits of precision, and perturbed by $\varepsilon \in [10^{-7}, 10^{-2}]$. We compare our dynamic filter with the semi-static filter of [5].

There was a slight difference in the overall running time between using the dynamic filter and using the semi-static filter. The time spent in the portion of the code performing arithmetic operations was only about twice more for the interval filter than for the semi-static filter, which is consistent with the results of Table 1. This is only about a third of the overall computation time. Neither filter made a mistake on the random cases, nor did the standard floating-point implementation. More interestingly,

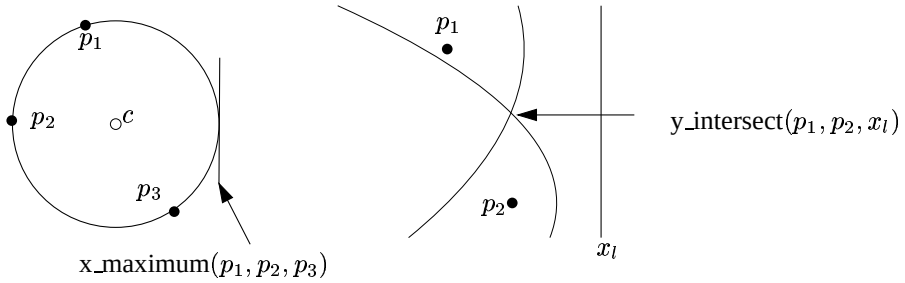

Figure 1: The constructions for the sweep algorithm. The predicates in the algorithm compare two y_intersect, or an input abscissa with a x_maximum, or two x_maximum values. The expressions computing these values are given in the text.

on perturbed grid or circle, we clearly demonstrate the efficiency of the interval filter, which rarely fails, whereas the semi-static filter shows the weakness of its bounds. For perturbed circular configurations, the interval filter fails consistently at least once only for perturbations smaller than 10^{-7}. We give in table 5 the number of failures of both filters on these two cases. Since we didn't use exact arithmetic, in case of unsafe comparisons, we used either the median value or one of the bounds of the interval to conclude. This is not robust, and the dashes indicate when the floating-point computation failed and the overall algorithm crashed because of inconsistencies. Interestingly, we saw that the median value of the interval may be a more stable approximation than the floating-point computation.

In the case of degenerate points, neither filter can of course detect the degeneracies, but they have similar running times, even though the filters always fail. The running time of the algorithm is therefore a function of the exact arithmetic used in case of failures of filters, which is outside the scope of this paper.

	circular $+\varepsilon$ 1000 points		grid $+\varepsilon$ 70×70 points	
	semi-static	dynamic	semi-static	dynamic
10^{-2}	11	0	20	0
10^{-3}	39	0	22	0
10^{-4}	43	0	41	0
10^{-5}	76	0	53	0
10^{-6}	106	0	–	8
10^{-7}	–	1	–	62

Table 5: Number of failures of the filters for the sweep algorithm for Voronoi on different almost degenerate distributions. Dashes mean that unsafe comparisons (when the filter fails but we still try to conclude) lead to corrupt data structure and failure of the algorithm.

Portability issues.

Our implementation of interval arithmetic contains some non-portable code for the adjustment of the IEEE754 rounding modes. There are mainly two ways to implement the rounding control. The first is to use appropriate library routines shipped with the compiler, if available. Unfortunately, there is no uniform interface for the rounding control, and function names tend to change with new compiler versions. Moreover, the reliability of the library functions is sometimes a problem. The second way is to directly use assembly code for manipulating the control word of the floating-point unit (FPU) in the processor. Although the programming in assembly code is technically demanding, it has the big advantage that it removes the dependency of the code from the used compiler version. In our implementation we use well-tested combinations of both methods. A general interface for the FPU access for some of the main Unix platforms (including Intel386, SPARC, MIPS, and alpha) will soon be available as a part of the LEDA extension package `Numbers`. The latest version of this package can be downloaded at `http://www.mpi-sb.mpg.de/~burnikel/Numbers.html`.

7 Conclusion

We have presented an effective interval technique for computing signs of determinants, which is a problem that frequently arises in computational geometry. In contrast to the straightforward application of interval arithmetic to Gaussian elimination such as in Algorithm 1, which only works for relatively small dimensions, our new Algorithm 2 can handle even very large-dimensional, fairly ill-conditioned matrices. Algorithm 2 is not more than one order of magnitude slower than the naive, inexact floating-point computation and is faster than all the exact methods. It remains an interesting open problem to design an efficient filter for large-dimensional sparse or structured matrices, that takes advantage of the structure of the matrix.

We have also investigated the general use of interval arithmetic in dynamic arithmetic filters for various low-dimensional geometric predicates. Interval arithmetic gives the user much more flexibility than static or semi-static error bounds, because divisions and square roots can be handled without greatly decreasing the probability of success of the filters. Using interval arithmetic it is neither necessary to restrict the input domain of the filters, as in the case of static filters, nor is it necessary to precompile the user program as in the case of semi-static filters. Our implementation of interval arithmetic is particularly efficient, because it avoids the expensive manipulation of the IEEE754 rounding mode preceding every interval operation, and is significantly faster than other dynamic filters.

For low-dimensional geometry, we have packaged our interval filter for the CGAL library.[7] Our package is available independently at `http://www-sop.inria.fr/prisme/personnel/pion/`. Experiments show that it rarely fails on non-degenerate instances that make the semi-static filter fail. Hence, we recommend interval arithmetic as the ultimate level of filter before resorting to efficient exact arithmetic. In most cases, we expect that resorting to exact arithmetic will not be needed. Should this be the case, however, several options are available depending on the type of operations used by the predicate. For rational operations, a general purpose bignum library

[7] `http://www.cs.uu.nl/CGAL/`

like GMP, CLN or packageLEDA integers, will suffice. The most general method we know of are the LEDA reals [6] which perform adaptive exact computation and handle arbitrary algebraic operations (by means of separation bounds).

References

[1] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec, Evaluating signs of determinants using single-precision arithmetic, Algorithmica 17 (1997) 111–132.

[2] J. Blömer, Computing sums of radicals in polynomial time, in: Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci. (1991) 670–677.

[3] H. Brnnimann, I. Emiris, V. Pan, and S. Pion, Sign detection in residue number systems, in: Theoret. Comput. Science 210:1 (1999) 173–197.

[4] H. Brnnimann and M. Yvinec, Efficient exact evaluation of signs of determinants, in: Proc. 13th Annu. ACM Sympos. Comput. Geom. (1997) 166–173. 1997.

[5] C. Burnikel, S. Funke, and M. Seel, Exact geometric predicates using cascaded computations, in: Proc. 14th Annu. ACM Sympos. Comput. Geom. (1998) 175–183.

[6] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig, Efficient Exact Geometric Computation Made Easy, in: Proc. 15th Annu. ACM Sympos. Comput. Geom. (1999) 341–350.

[7] C. Burnikel, Exact Computation of Voronoi Diagrams and Line Segment Intersections, Ph.D. Thesis (Universität des Saarlandes, Germany, 1996).

[8] K. L. Clarkson, Safe and effective determinant evaluation, in: Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci., (1992) 387–395.

[9] O. Devillers and F. P. Preparata, A probabilistic analysis of the power of arithmetic filters, Technical Report CS-96-27 (Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996).

[10] A. Edalat and A. Lieutier, Foundation of a computable solid modeling, in: ACM SIGGRAPH Symp. on Solid Modeling Ann Arbor (1999).

[11] S. Fortune, A sweepline algorithm for Voronoi diagrams, in: Algorithmica 2:2 (1987) 153–174.

[12] S. Fortune, Voronoi diagrams and Delaunay triangulations, in: D.-Z. Du and F. K. Hwang, eds., Computing in Euclidean Geometry, volume 1 (Lecture Notes Series on Computing, World Scientific, Singapore, 1992) 225–265.

[13] S. Fortune and C. J. Van Wyk, Efficient exact arithmetic for computational geometry, in: Proc. 9th Annu. ACM Sympos. Comput. Geom. (1993) 163–172.

[14] S. Fortune and C. J. Van Wyk, Static analysis yields efficient exact integer arithmetic for computational geometry, in: ACM Trans. Graph. 15:3 (1996) 223–248.

[15] G. H. Golub and C. F. van Loan, Matrix Computations (Johns Hopkins University Press, third edition, 1996).

[16] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, C++ Toolbox for Verified Computing, (Springer, 1995).

[17] IEEE, IEEE standard 754-1985 for binary floating–point arithmetic, reprinted in: SIGPLAN Notices 22:2 (1987) 9–25.

[18] K. Mehlhorn and S. Näher, The implementation of geometric algorithms, in: Proc. 13th World Computer Congress IFIP94, volume 1 (Elsevier Science B.V. North-Holland, Amsterdam, 1994) 223–231.

[19] R. E. Moore, Interval Analysis (Prentice-Hall, 1966).

[20] A. Neumaier, Interval Methods for Systems of Equations, (Cambridge University Press, 1990).

[21] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, in: Disc. Comput. Geom. 18 (1997) 305–363.

[22] C. K. Yap and T. Dubé, The exact computation paradigm, in: D.-Z. Du and F. K. Hwang, eds., Computing in Euclidean Geometry (Lecture Notes Series on Computing, volume 1, World Scientific, Singapore, second edition, 1995) 452–492.

[23] C. K. Yap, Robust geometric computations, in: J. E. Goodman and J. O'Rourke, ed., Handbook of Discrete and Computational Geometry (CRC Press LLC, Boca Raton, FL, 1997) 653–668.