# Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry

Hervé Brönnimann, Christoph Burnikel, Sylvain Pion

# Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry*

Hervé Brönnimann† Christoph Burnikel‡ Sylvain Pion†

**Abstract**

We discuss interval techniques for speeding 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. Furthermore we show how to use our interval techniques for exact linear optimization problems of low dimension as they arise in geometric computing. 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. There are numerous approaches to get the problem under control, of which the immediate solution of exact computation stands out because of its generality. For faster yet exact computation, arithmetic filters were proposed in [9, 15] and showed to be efficient both in practice [10, 18] and in theory [7].

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. Static filters are restricted to integral expressions of small bounded depth and, which is even worse, require that good upper bounds on the input variables are known in advance. Semi-static filters remedy most of the mentioned 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 use semi-static filters the complete structure of a computation has to be preprocessed in advance. Dynamic filtering on the other hand is often considered too inefficient for use in computational geometry. First advances have been made to incorporate dynamic filters into the LEDA reals [5].

In this paper, we propose to use *interval analysis* [16, 17, 13] for more efficient dynamic filters. The technique is based on carefully engineered interval arithmetic. It is very simple to use and yields the most flexible dynamic floating-point filters we know: divisions can be handled as well as square roots and hence the technique is not limited to rational expressions. With the IEEE 754 standard for floating point computations [14], 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, being roughly 3-8 times slower than floating-point evaluation. Our implementation of interval arithmetic is heavily influenced by that of 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 [6, 1, 3], or using general solutions such as exact integer arithmetic [9] and modular arithmetic [2]. For $d \times d$ determinants, the complexities range from $O(d^3 \log d)$ to $O(d^4 \log d)$ 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. In this paper we engineer a fast floating-point filter for computing the sign of a determinant in time $O(d^3)$ with a small constant. Our

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

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

‡burnikel@mpi-sb.mpg.de. Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany.

filter fails only for matrices which are singular or nearly singular. There is also 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 cannot be used here.

Another class of problems that we consider are geometric optimization problems solvable by linear programming. Here the central predicate is computing the sign of a dot product over vectors whose coordinates are determined by matrix operations, including matrix inversion. In [11], Gärtner describes an exact implementation of the simplex algorithm using a semi-static floating-point filter for better efficiency. However, Gärtner needs to compute the inverses of the base matrices with exact arithmetic. In [13] it is shown how to compute a verified inclusion for the solution of a linear system $Ax = b$ and the method is applied to a partially correct implementation of the simplex algorithm. We present another non-iterative filter method having the advantage that one can state an explicit condition under which the method is applicable. Here we can avoid to compute an interval inclusion for the inverse $A^{-1}$.

There are two intrinsic limitations to the use of interval arithmetic. First, interval arithmetic may fail to detect degenerate instances because of the roundoff errors. When the operations are performed exactly, the interval is reduced to a point and this certifies that there is indeed a degeneracy. However, this only happens when the bitlength of the input data is small with respect to the machine precision. Second, although the computed intervals are optimal for every single operation, there can be a significant overestimation of the error for a cascaded sequence of operations. 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 degree is constant, i.e., independent of the dimension of the matrix. This means that our filter can come into effect for matrices of arbitrary dimension. The same is true for the predicates used in the simplex algorithm.

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. We then discuss a new implementation of interval arithmetic that relies on the rounding modes of the IEEE 754 standard. In section 3 we introduce the basic notions of interval analysis and give an overview of the efficient methods to obtain verified inclusions. Here we introduce a heuristic measure of the effectiveness of interval analysis for a given expression $\mathcal{E}$. In section 4 we present our filter for computing the sign of a determinant and in section 5 we sketch an implementation of the simplex algorithm that uses interval arithmetic. In section 6 the approach is validated experimentally. Beside the mentioned applications we also consider other geometric predicates, such as those encountered in optimization or Delaunay sweep algorithms.

## 2 Arithmetic filters

Filters are used when determining the sign of a fixed expression $\mathcal{E} = \mathcal{E}(x_1, \ldots, x_n)$. They allow to evaluate the sign in a robust manner, while being far quicker than the complete exact evaluation, in most cases. A filter never returns a wrong answer, but may fail to return a meaningful answer at all (in this case returning NO_IDEA).

We will focus on single precision[1] floating point filters, because they have a speed comparable to the simple floating point evaluation. We distinguish mainly three kinds of such filters, described below.

**Fully static:** An upper bound $|x_i| \leq X_i$ is known for each $i$, and $\mathcal{E}$ contains only $+, -, \cdot, \sqrt{}$. Then $E$ is computed such that the error on computing $|\mathcal{E}|$ is bounded by $E$ for all inputs. For a particular input, the filter fails if $\mathcal{E} \leq E$, otherwise the sign of $\mathcal{E}$ is known safely.

**Semi-static:** Sometimes it is impossible to specify a good bound on the entries, but there is a simple formula $E = E(x_1, \ldots, x_n)$ having a structure similar to $\mathcal{E}$ that gives a valid error bound for a particular input, even when $E$ is evaluated with single precision. $E$ is computed dynamically, and the filter fails if $|\mathcal{E}| \leq E$; otherwise the sign of $\mathcal{E}$ is known safely.

**Dynamic:** the computation of $E$ is carried along with the computation of $\mathcal{E}$. Typically, for each operation of $\mathcal{E}$, a simple rule determines the error bound for the result of that operation based on the operands and error bound on them.

Exact computation can be considered the last of filters, whcih never fails. The cost of the total computation can then be expressed as a combination of the cost of the different filters multiplied by their conditional probability of success.

Static filters are implemented for instance in LN [10], semi-static in [1, 4], and dynamic filters in Expr [19] and LEDA [5]. Also Schewchuk computes $\mathcal{E}$, then the error, then the second order error, etc., until the sign can be safely determined; this can be considered as a dynamic filter according to our description. A static filter does the floating point evaluation of $\mathcal{E}$ plus one extra comparison, whose running time is usually neglectable.

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

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.

We now describe in detail a particular filter, which is dynamic, based on interval arithmetic.

## 3 Interval arithmetic

The major tool used within our filter is *interval arithmetic*. The use of interval arithmetic in the context of matrix operations was originally proposed by Moore [16] and further promoted through a research group directed by Kulisch; see [13] for a recent survey of the available computational methods. In [13], interval arithmetic is successfully applied to many basic computation tasks in numerical linear and nonlinear algebra. However, the problem considered in the present paper seems to have been overlooked so far. One reason might be that the computation of determinants is, unlike the computation of eigenvalues of symmetric matrices, not easily tractable by the standard interval method of using Brouwer's fix-point theorem [17].

Interval arithmetic deals with *intervals* $[x] = [\underline{x}, \overline{x}]$ of real numbers. 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. In fact, our methods can be applied when the input matrix $A$ has interval coefficients, but in our applications the coefficients of $A$ are given exactly. Note that for interval–type input matrices the determinant does not necessarily have a unique sign, hence in this case the problem is not always well–defined.

The basic interval operations are defined essentially as in [13]. Namely, if both operands $[x] = [\underline{x}, \overline{x}]$, $[y] = [\underline{y}, \overline{y}]$ are finite intervals we set

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

If one of the intervals $[x]$, $[y]$ is infinite, we set the resulting interval to $\mathbb{R} = [-\infty, \infty]$. 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 $\bigtriangledown\underline{x}$ and $\bigtriangleup\overline{x}$ are the numbers in $\mathbb{F}$ next to $\underline{x}$ and $\overline{x}$ when rounding downwards (resp. upwards). Then the approximate floating-point interval operations are given by

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

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 $f([x])$. Let $\mathcal{E}$ be an arithmetic expression over the operations $\{+, -, \cdot, /, \sqrt{\ }\}$. If the expression is applied to intervals $[x_1], \ldots, [x_n]$ instead of real numbers we denote the resulting interval expression by $[\mathcal{E}]$ for clarity. For the rest of the paper we assume that interval expressions are evaluated using the approximate interval operations $\oplus, \ominus, \odot, \oslash$ and $\mathrm{sqrt}_\diamond([x]) = \diamond([x]^{1/2})$.

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[2]. 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 number of uncertain bits of $[\mathcal{E}]$ 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 $\mathrm{Ideg}(\mathcal{E}) = 0$ and inductively, if $\mathcal{E}$ is computed from expressions $\mathcal{X}$, $\mathcal{Y}$ by the operations $\{+, -, *, /, \sqrt{\ }\}$ we set

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

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

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

**Lemma 1** *The interval solution of a triangular system $T \cdot [x] = b$ with exact input data has degree $d$.*

**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 $x_0 = b_0/t_{0,0}$ which has degree 1. Let for $d > 0$ $x_0, \ldots, x_{d-1}$ have degree $\leq d$ and $\mathrm{Ideg}(x_{d-1}) = d$. From the formula

$$x_d = b_d/l_{d,d} - \sum_{j=0}^{d-1} x_j(l_{d,j}/l_{d,d})$$

we see that since $\mathrm{Ideg}(l_{d,j}/l_{d,d}) = 1$ each product $x_j(l_{d,j}/l_{d,d})$ has degree $\leq d+1$. Because the latter term has degree $d+1$ for $j = d$, the whole sum has degree $d+1$. 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, even if $A$ is tridiagonal. Another simple lemma is that Gauss elimination is of degree $2d$.

Let us stress again that our notion of 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 degree or else if the problem has small dimension[3]. Nevertheless, our degree is logarithmically related to the index of [4] (if we do not count additions), and this index gives a bound on the relative error of a computation.

As we just said, many of the most important computations in linear algebra have non-constant degree. How can we save the interval method? One powerful approach is the following. Reformulate the problem such that the required solution is expressed as the limit of an iteration process $x \leftarrow f(x)$ where $f(x)$ has small 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 fixpoint theorem to prove that the iteration converges to a solution and hence the solution is contained in $I$. This is an *a posteriori* method since one only has to consider the errors made in the computation of $f$, for every single step of the iteration. This approach works well for many of the basic problems in numerical mathematics, including solutions of linear and nonlinear systems, global optimization and automatic differentiation [13].

Unfortunately, the fix-point method does not seem to work so smoothly for the computation of a determinant. Why? Well, 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 $\lambda'$ of $A$, we could further use a Newton iteration that converges to the desired solution of $(A - \lambda I)x = 0$ as in [13]. 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 $A$. Our solution is quite different from the method mentioned before.

[3]For problems of large degree in large dimensions, interval arithmetic can still be useful if the input data has an appropriate (sparse) structure.

## 4 Computing the sign of a determinant

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

- Exact integer or floating-point arithmetic
- Exact modular arithmetic [2]
- Clarkson's re–orthogonalization method [6, 3]
- The lattice method [1, 3]

In order to apply some of these methods, it is necessary to make the matrix entries integral by multiplying the matrix with a large enough power of 2. Note that this scaling does not change the determinant's sign, but imposes severe restrictions on the input matrix. Namely, its coefficients must be exactly representable integers in $\mathbb{F}$. The first method has a running time of $d^3M(d)$, $M(d)$ being the time required to compute the product of two numbers of word-length $d$ (words in $\mathbb{F}$ have length 1). The modular variant runs in time $O(d^4 \log(d))$, Clarkson's algorithms runs in time $O(d^3 \log(d))$ [3]. The last algorithm has exponential worst-case running time but is reported in [3] to take only time $O(d^3)$ in practice, like Clarkson's algorithm. It should be noted that in the general case, the four methods above all require multi-precision, hence there is a potentially large constant hidden in these asymptotic bounds. If the entries are substantially smaller than the mazimum integer representable in $\mathbb{F}$, however, Clarkson's algorithm incurs little overhead on top of direct floating point computation. If the matrix is reasonably orthogonal, the running time of the last two methods is close to $O(d^3)$, these algorithms are said to be adaptive; they behave like a filter. The first two methods, however, handle all cases similarly and are not adaptive.

In this paper 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 effective also if the input is not representable in $\mathbb{F}^{d,d}$. The filters fails only for matrices that are singular or nearly singular.

### 4.1 A posteriori method

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. The determinant of $P$ is $\pm 1$ and can be computed without rounding error, and $\det(L)$ is 1 since the diagonal elements of $L$ are all equal to 1. So the product of the diagonal elements $u_{i,i}$ of $U$ gives an approximation of $\pm\det(A)$. For well-conditionend matrices this approximation is usually quite reliable. 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 $A^{-1} \approx U^{-1}L^{-1}P$. Since 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 given 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 regular 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 $D : t \mapsto \det(I + t \cdot F), t \in [0,1]$. $D$ is continuous and $D(0) = 1$. Since all matrices $I + t \cdot F, t \in [0,1]$ are regular by the argumentation above, $D(t)$ is always nonzero. By continuity of $D$, this means that $D(1) = \det(I+F) > 0$. ∎

All we have to do now is to check whether the defect matrix $E = I - B * A$ has norm $< 1$ for some matrix norm of our choice. To this end, evaluate $[I - B \cdot A] = ([e_{i,j}])_{i,j}$ by interval arithmetic. 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 $\mathrm{sign}(\det(A))$ is as follows. Note that the algorithm either returns $+1$, $-1$, or the string NO_IDEA (if the filter fails).

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

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

Remarks:

- 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.
- If Step 1 was correctly completed, Step 2 can only fail because of exponent overflow.
- 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 $\mathrm{sign}(\det(B))$ has to be computable exactly.

**Lemma 3** *Algorithm 1 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. Computing the product of a full matrix with a triangular matrix takes $d^3/2 + O(d^2)$ operations. All other computations require only $O(d^2)$ operations. ∎

Let us now give a rough estimate for the running time of algorithm 1 with respect to the unfiltered evaluation of $\mathrm{sign}(\det(A)) \approx \mathrm{sign}(\det(P)) \prod_i \mathrm{sign}(u_{i,i})$ ("naive algorithm"). Hammer et al. [13] 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 $3 - 4$ is realistic, though. Our measurments showed that algorithm 1 takes in fact $9 - 12$ times longer than the naive algorithm.

### 4.2 Naive interval arithmetic

Algorithm 1 is applicable even for very large dimensions; it was successfully tested up to $d = 800$. On the other hand, many applications, e.g. in computational geometry, are small–dimensional ($d \leq 25$). For small dimensions, an overhead factor of about 10 is not always better than the overhead that we get using one of the exact algorithms. However, remember that for those cases the straight evaluation of the determinant in interval arithmetic is possible. Hence we propose the following alternative algorithm.

**Algorithm 2** *(returns the sign of det(A), if possible; for small dimensions)*

1. *Compute $PA = [L] \cdot [U]$ in interval arithmetic.*
2. *If one of the intervals $[r_{i,i}]$ on the diagonal of $[U]$ contains zero, return NO_IDEA. Otherwise, return $sign(det(P)) \cdot \prod_i sign([r_{i,i}])$ where $sign([r_{i,i}])$ is the sign of any number in $[r_{i,i}]$.*

Remarks:

- Step 1 can always be completed, since the division by a "pivot" interval containing 0 is a regular operation that results in $[-\infty, \infty]$.
- The lower triangular part $[L]$ does not have to be computed explicitly, since it is not needed in the sign calculation.

The algorithm is best applied to matrices of small dimension. However, for large matrices with a special structure, like certain sparse matrices as well as block matrices of small block size, it might still be useful. If it returns a result, it always outperforms algorithm 1. We measured that algorithm 1 is only at most 2.6 times slower than with naive floating point calculation, for random entries and dimensions $5 - 40$.

On the other hand, algorithm 1 has a higher probability of success since the computed intervals are smaller. See section 6 for detailed measurements.

**Lemma 4** *Algorithm 2 takes at most $d^3/3 + O(d^2)$ interval operations.*

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

**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 $\sigma_i$ can differ by at most $\|F\|_2$, see Corollary 8.6.2 of [12]. This implies that

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

If the coefficients of $F$ are all bounded by $\varphi$, we get $\|F\|_2 < d\varphi$ and hence $|1 - \det(I + F)| \ll d^2\varphi$. From Lemma 5 and from $\det(A) = \det(B \cdot A)/\det(B)$ we get an enclosure for $\det(A)$, evaluating $\det(B) = \prod_i b_{i,i}$ with interval arithmetic.

### 4.3 Discussion

Our new algorithm 1 is an effective mechanism to filter out "easy" cases of the sign determination. The algorithm is attractive because of its numerical quality (it fails only for nearly singular matrices) and because it is not restricted in the dimension (other filters work only for small dimensions). If the input matrix $A$ has arbitrary floating–point entries, it is often much more efficient than the exact methods.

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. Note that interal arithmetic may also be exact if the entries are small compared to the machine precision. Second, the interval method of algorithm 1 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 2 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 large sparse matrices.

## 5 Filters for the simplex algorithm

In a paper to appear in SODA'98 [11], Gärtner discusses an exact implementation of the revised simplex algorithm. The optimization problem he considers is in the standard form

$$\begin{array}{lll} \text{(LP)} & \text{maximize} & z = c^T \cdot x \\ & \text{subject to} & Ax = b \\ & \text{and} & x \geq 0, \end{array}$$

where $A$ is an $m \times n$ matrix, $c, x$ are $n$-vectors, $b$ is an $m$-vector, and we assume $m \ll n$. The latter assumption is realistic for geometric optimization problems such as the smallest enclosing annulus of $n$ points in $\mathbb{R}^d$ ($m = d + 2$ variables, $n$ constraints), or the largest disk in the kernel of a simple $n$-vertex polygon ($m = 3$).

One iteration of the revised simplex algorithm consists of three parts, the *pricing step* where an index $i$ is chosen that enters the basis, the *ratio test* where an index $j$ is chosen that leaves the basis to be replaced by $i$,

and the *update step* where the auxiliary data structures are adapted to the new basis.

Gärtner achieves the correctness of his implementation by using exact computation for the critical steps of the simplex algorithm. These steps are (1) checking the consistency of the chosen candidate for the index $i$ in the pricing step, and (2) the ratio test. To achieve efficiency, Gärtner computes a candidate for the index $i$ by straightforward floating point computation. To make this approach even more appealing, Gärtner uses the pricing strategy *partial reduced cost pricing* going back to Dantzig and Clarkson. With this strategy, the pricing step dominates the running time of the simplex algorithm and hence the fraction of exact computation is minimized, taking advantage of $m \ll n$. Moreover, this pricing strategy allows to use a semi-static floating point filter to reduce the time penalty of exact computation.

However, Gärtner always computes the inverse of the base matrix by exact arithmetic. This is because in the computation of the inverse, as Gärtner puts it, "the error bounds obtained from interval arithmetic are typically a gross overestimate, making them too large to be useful even for small values of $m$." In fact, it does not make much sense to compute an interval inclusion of the base inverse using direct techniques such as Gauss-Jordan elimination. Again, the trick is to reduce the arithmetic degree, i.e. the interval degree of the computation. Here one can choose among the two basic methods mentioned in the last section, which are (1) the use of appropriate fixed point theorems and (2) a posteriori error analysis. Both methods use interval arithmetic as described in section 3. The second option has already been discussed extensively in the literature on interval computation [13] and has shown to be effective. In this paper we sketch the second method that relies on the following lemma.

**Lemma 6** *Let $A, B \in \mathbb{R}^{d,d}$ where $B$ is an approximate inverse of $A$ such that $\|I - BA\| < 1$. Then for any vectors $x, b \in \mathbb{R}^d$ the approximate solution $y = Bx$ of the system $Ax = b$ satisfies the estimate*

$$|y - A^{-1}b| \leq \frac{1}{1 - \|I - BA\|} |B(Ay - b)|.$$

*Here $\|\cdot\|$ is the matrix norm corresponding to the vector norm $|\cdot|$.*

**Proof:** From

$$|y - A^{-1}b| = |(BA)^{-1}B(Ay - b)| \leq \|(BA)^{-1}\| |B(Ay - b)|$$

the statement follows by applying Lemma 2 to the matrix $F = I - BA$. ∎

The previous lemma allows the following strategy for finding an inclusion of the solution of $Ax = b$. First, compute an approximate floating-point inverse $B$ of $A$ and an approximate solution $y$ of $Ax = b$. Then, compute an interval inclusion for the defect $[I - D] = [I - B \cdot A]$ and the quantity $[z] = [B(Ay - b)]$. If $\|[I - D]\| < 1$, we get the error bound

$$|y - A^{-1}b| \leq \frac{1}{1 - \|[I - D]\|} [z].$$

Although the revised simplex method can be implemented using the method sketched above, it has to be refined for better efficiency. The expensive step here is the computation of the defect $[I - D]$ which takes $d^3 + O(d^2)$ operations.[4] Since the update of a basis matrix $M$ is done by $M' = M \cdot \Lambda$ with a matrix $\Lambda$ of the form

$$\Lambda = \begin{pmatrix} 1 & & & \lambda_1 & & & \\ & \ddots & & \vdots & & & \\ & & 1 & \lambda_{i-1} & & & \\ & & & \lambda_i & & & \\ & & & \lambda_{i+1} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & \lambda_m & & & 1 \end{pmatrix},$$

we can update the defect matrix by $[D'] = [\Lambda^{-1}][D][\Lambda]$. It is easy to see that the update needs only $O(d^2)$ interval operations and the interval degree grows by at most 2. After some number of iterations (say, after 20 iterations) it is necessary to recompute the defect interval $[D]$ from scratch.

Let us now discuss why it is not a good idea to compute an inclusion for the inverse of the base matrix $M$. First, doing this with Gaussian elimination has degree $2d$ and all other methods either have the same degree or need more operations. Second, the use an inclusion $[M^{-1}]$ forbids the use of a semi-static filter in the pricing step and in the ratio test. This is because a semi-static filter needs the input quantities rounded correctly to the next floating point value. By the method sketched above, we can first start with a semi-static filter similar to that used by Gärtner. Note that the intervals obtained with this method are much smaller than if we used interval arithmetic directly in the Gaussian elimination. Only when this filter fails, we use dynamically computed intervals. In practice, we do not expect to resort to exact computation except for degenerate cases. We therefore hope to keep the cost almost the same as that of the floating point implementation.

[4] In the computation of the defect interval we do not strictly need our packaged interval arithmetic since it is enough to evaluate $BA$ once rounding towards $+\infty$ and once rounding towards $-\infty$.

## 6 Experimental Validation

We experiment our filter in a variety of settings encountered by geometric algorithms. In this paper, we have not investigated the practical efficiency of our filters for solving linear problems.

**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.

We evaluate the semi-static filters given in [4], the semi-static error bound being evaluated once only for the entire predicate in "Semi-static," and once for each operation in "FpFilter."

Our dynamic filters "OutFilter" and "InFilter" implement interval arithmetic using IEEE standard 754 rounding towards $+\infty$. Note that, since we maintain the negative of the lower interval bounds, we never need to round towards $-\infty$. Like for semi-static filters, the rounding mode is manually set once only outside the predicate in the variant "OutFilter," and for every single operations inside the interval class in the variant "InFilter."

The dynamic filter types "AbsFilter" and "RelFilter" use standard error bounds to dynamically compute absolute errors and relative errors, respectively.

The considered predicates are the standard orientation and insphere predicates, and a specific incircle predicate for the Voronoi diagram of line segments that asks whether a point in the circle circumscribed to two lines and a point site. The latter predicate involves three square root operations but no divisions. The benchmarks of table 2 were made on a Ultra Sparc 200 MHz, with the GNU compiler using the flag -O2.

| | Insphere 3D | Orient 3D | Orient 2D | Incircle 2D (llp_p) |
|---|---|---|---|---|
| Semi-static | 2.28 | 1.63 | 1.45 | 1.58 |
| FpFilter | 3.36 | 2.55 | 1.77 | 2.17 |
| OutFilter | 5.15 | 2.98 | 1.92 | 3.38 |
| InFilter | 10.58 | 5.85 | 3.13 | 4.89 |
| AbsFilter | 20.00 | 10.68 | 5.38 | 7.42 |
| RelFilter | 94.00 | 102.5 | 69.70 | 9.23 |

Table 2: Running time overhead caused by the use of various filters, compared to the the static filter (pure floating-point computation). The time taken by other computations in case of filter failure is not taken into account.

As our experiments show, the variant "OutFilter" is always the fastest. Hence in all other experiments, we use only this filter variant. Table 2 documents only differences in the running times, without accounting for the other computations in case of failure. Moreover, the filters have different probabilities of success. For a fair comparaison, one needs to evaluate them within some geometric algorithm. This is the topic of both sections below.

**Computing determinants.** We investigate the efficacy of our interval filter for computing signs of nearly degenerate determinants. Our test determinants have all floating point entries near to 1 but with a random perturbation of size $\leq 2^{-p}$, i.e., after the $p$th bit. A meaningful parameter to investigate the quality of the filter is the average value $k$ of the parameter $p$ such that failure rate gets above 50%. In table 1 we display this number $k$, first for the naive method and then for the a posteriori method. A dashed entry means "fails always."

**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 [8]. The predicates involve orientation tests, comparing the ordinates of a point and of an intersection of two parabolae, 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.

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 on 53 bits between $\frac{1}{2}$ and 1, and perturbed by $\varepsilon \in [10^{-7}, 10^{-2}]$. We compare our dynamic filter with the semi-static filter of [4].

There was no great difference between the running times of our implementations of a dynamic and semi-static filter. Over all the course of the algorithm, interval arithmetic was only twice/three times slower then the semi-static filter. Neither filter made a mistake on the random cases, nor did the standard floating point implementation. More interestingly, on perturbed grid or circle, we clearly demonstrate the efficiency of the interval filter, which rarely fails, whereas the static filter shows the weakness of its bounds. For perturbed circular configurations, the interval filter needs perturbation by $10^{-10}$ to fail at least once consistently. We give in table 3 the number of failures of both filters on these two cases. Since we didn't use exact arithmetic, the dashes indicate when the floating point computation failed. Interestingly, we see 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.

| method, n | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 20 | 24 | 28 | 32 | 44 | 52 | 68 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| naive | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 37 | 37 | 32 | 28 | 24 | 21 | 11 | 2 | – |
| a posteriori | 46 | 45 | 45 | 44 | 43 | 43 | 43 | 42 | 42 | 42 | 40 | 40 | 39 | 39 | 39 | 38 | 37 | 36 |

Table 1: The number of perturbed bits for which the interval filters fail for at least 50% of the cases.

| | circular $+\varepsilon$ 1000 points | | grid $+\varepsilon$ 70×70 points | |
|---|---|---|---|---|
| | static | dynamic | 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 3: Number of failures for the filters on different almost degenerate distributions.

## 7 Conclusions

The best prior art proposes static filters to low-dimensional primitives dealing only with integer entries [1, 3, 6, 10], or to rational 2d and 3d primitives when dealing with floating-point entries [18]. Our solution is powerful as it avoids the overestimation of errors induced by static bounds and is efficient, i.e., always less than an order of magnitude slower than the straightforward floating-point implementation. Unlike previous filters, it can handle square roots and divisions. Square roots are needed for instance in computing the Voronoi diagram of a set of points or line segments, and divisions are needed in the efficient computation of the sign of a large determinant. For the latter problem, our filter is the first that works for arbitrary dimensions.

For low-dimensional geometry, we have packaged this filter for the CGAL library. Experiments show that it is about as slow as the semi-static filter when used in a predicate and with a precompiler such as [4], but 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.

## References


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

[2] H. Bronnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[3] H. Bronnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, 1997.

[4] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computations. In *Submitted to this conference*. ACM, 1998.

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

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

[7] 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.

[8] S. Fortune. Voronoi diagrams and Delaunay triangulations. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 225–265. World Scientific, Singapore, 2 edition, 1992.

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

[10] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometr y. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[11] B. Gärtner. Exact arithmetic at low cost - a case study in linear programming, proc. 9th annual acm-siam symp. on discrete algorithms. In *Proc. 9th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA), pages 157–166, 1998.*

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

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

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

[15] K. Mehlhorn and St. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[16] R.E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[17] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

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

[19] C. K. Yap and T. Dubé. The exact computation paradigm. In *Computing in Euclidean Geometry*, D.-Z. Du and F. K. Hwang, eds., Lecture Notes Series on Computing, volume 1, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.